Facultad
de
Ciencias

APLICACIÓN DE TÉCNICAS DE DEEP LEARNING A LA CLASIFICACIÓN DE *GASTROPODA CONIDAE*

(DEEP LEARNING APPLICATIONS IN *GASTROPODA CONIDAE* CLASSIFICATION)

Trabajo de Fin de Grado
para acceder al

GRADO EN FÍSICA

Autor: Luis Moro Carrera

Director: Lara Lloret Iglesias

Co-Director: Fernando Aguilar Gómez

Julio - 2020

Agradecimientos

Quiero comenzar agradeciendo a Lara Lloret y Fernando Aguilar por todo el tiempo que me han dedicado. Data science es un campo muy novedoso y ellos han invertido un gran esfuerzo personal para enseñarme multitud de conceptos y modelos. Gracias por haberme dado esta oportunidad y por ser unos guías maravillosos.

A Ignacio Heredia, por toda su paciencia explicándome los entresijos de Python y los Dockers.

A Aida Palacio, por solucionar cada dificultad con el hardware durante todo el trabajo.

Al grupo de Computación Avanzada y e-Ciencia del Instituto de Física de Cantabria (IFCA-CSIC-UC) por todos los recursos cedidos.

Gracias a mi familia por creer en mi todos estos años. Pese a ser tantos en casa, nunca me ha faltado de nada y siempre os habéis preocupado por mi bienestar. Gracias por arroparme con todo el estrés de hacer las dos carreras al mismo tiempo.

A mis amigos, por entender mis largas desapariciones durante los exámenes y trabajos.

Resumen

El primer objetivo de este proyecto consiste en aprender a usar Python y sus librerías de aprendizaje profundo. Con este fin, el IFCA me ha cedido acceso a la página de Moodle del Máster en Ciencia de Datos. He tenido que terminar todas las tareas de Python de la asignatura "Introducción a los datos masivos" antes de empezar el proyecto real.

Después he pasado a las redes convolucionales. He formado parte de un proyecto real con el IFCA y el Museo Nacional de Ciencias Naturales donde he tenido que entrenar una red neuronal para que distinga entre 83 especies distintas de *Gastropoda Conidae*. Se trata de una familia de moluscos gasterópodos vulgarmente conocidos como Conus. Cada una de las especies de esta familia posee una concha con un patrón característico. Para lograr distinguirlos, he tenido que modificar la arquitectura de un Docker llamado "deep.oc.image-classification-tf." Es un Docker desarrollado por el grupo de e-Ciencia del IFCA. Se usaba originalmente para distinguir entre clases de objetos cotidianos. Tomando ventaja de que las redes convolucionales abstraen figuras geométricas similares en ambos casos, he podido reutilizar algunas de sus capas. He tenido que reentrenar las densas. También he tenido que desarrollar programas en Python para preprocesar los datos y analizar los resultados. He obtenido una precisión del 92.9% en los datos de validación.

Finalmente, he modificado la red para que pueda recibir información adicional sobre la localización geográfica de los Conus. He reentrenado con imágenes y localización como datos de entrada y lo he probado. Tras intentar distintas estructuras para las capas densas, he escogido una capa densa oculta que retorna un vector de 1024 parámetros. Esto ha hecho aumentar la precisión de la red hasta un 95.4%, demostrando que este tipo de información adicional mejora la clasificación.

Palabras clave: inteligencia artificial, aprendizaje automático, aprendizaje profundo, Python, Keras

Abstract

The first goal of this project consists in learning how to properly use Python and its deep learning libraries. In order to get familiar with Keras environment, IFCA has provided me an account with access to the Data Science Master homepage. I had to complete all of the "Introduction to massive data" tasks prior to starting the real project.

Then I moved to convolutional networks. I joined a real deep learning project with IFCA and National Museum of Natural Sciences where I had to train a neural network in order to distinguish between 83 different species of *Gastropoda Conidae*. It is a taxonomic family of marine gastropod molluscs commonly known as Conus. Each Conus species possesses a unique shell pattern. Aiming to classify them, I had to modify the architecture of a Docker called deep-oc-image-classification-tf. It was developed by IFCA e-science research group. It was originally used to distinguish different classes of all kind of everyday objects. Taking advantage of the similar geometric abstraction among Conus classes and image-tf, I was able to reuse some of the convolutional layers. I had to retrain the dense ones. I also had to develpop some code for data preprocessing and result analysis. I obtained an accuracy of 92.9% on validation data.

FInally, I modified the net so that it could receive extra information about the Conus sample geographical location. Then I retrained it with image and location data and tested it. After trying different kind of dense layer structures, I end up choosing a 1024 parameter hidden dense layer. It increased the net accuracy on validation data to 95.4%, showing that this kind of additional data improves the classification.

Key words: artificial intelligence, machine learning, deep learning, Python, Keras

1. Introducción

Este trabajo pertenece al ámbito de la inteligencia artificial. Se emplean técnicas de aprendizaje profundo para resolver un problema real en el Instituto de Física de Cantabria. Es una colaboración con el Museo Nacional de Ciencias Naturales. Necesitaban una forma de etiquetar por especie distintos ejemplares de *Gastropoda Conidae* utilizando sus fotografías. Se trata de una familia de moluscos gasterópodos cuyas especies poseen conchas con patrones que las caracterizan. Coloquialmente se cononcen como Conus. En la sección 1.1 se detallan más aspectos de estas muestras y se enseñan algunos ejemplos. Se ha desarrollado una red capaz de distinguir dichas especies mediante imagen. Después, se ha dado la opción de añadir la localización geográfica donde se encontraron para mejorar la precisión en su identificación.

De forma similar, esta tecnología tiene diversas aplicaciones en física. Destaca en problemas de clasificación cuando se están manejando grandes cantidades de datos que excederían ampliamente lo que se puede manipular de forma supervisada por una persona.
Durante los últimos años se ha convertido en una herramienta habitual en prácticamente cualquier campo de investigación. Un ejemplo de ello es la física de partículas. El aprendizaje profundo se emplea en la identificación de jets de quarks, gluones y bosones cuando se tiene una gran cantidad de eventos para analizar. Este es el caso del LHC, como se puede leer en la fuente [1] de bibliografía. Es necesario disponer de una herramienta que permita cubrir tareas de clasificación y discriminación de sucesos relevantes cuando se tienen tales volúmenes de datos.

Lo relevante de este Trabajo Fin de Grado es por tanto entender los principios teóricos del aprendizaje profundo y cómo emplearlo de forma experimental. El objeto de estudio (los gastropoda conidae) es un pretexto para aprender a utilizar una herramienta fundamental en la física de hoy en día.

En esta sección introductoria vamos a explicar varios puntos. En la sección 1.1, se introducirán las muestras con las que vamos a trabajar. En la sección 1.2, se explicará por qué hasta la última década no había sido posible desarrollar este tipo de técnicas y cómo el desarrollo del hardware lo ha facilitado. En la sección 1.3 se contextualiza la Inteligencia Artificial y el aprendizaje profundo, que es la rama a la que pertenece este trabajo. En las sección 1.4 y 1.5 se explica el funcionamiento de las redes neuronales en las que se basa el aprendizaje profundo. Por último, en las secciones 1.6 y 1.7, se explica cómo se analizan los resultados obtenidos con técnicas de aprendizaje profundo y cuáles son las principales dificultades que pueden surgir al aplicarlas.

1.1 Muestras de Conus

Las muestras a analizar en este trabajo son imágenes de 83 especies distintas de la familia *Gastropoda Conidae,* llamada comunmente Conus o caracoles marinos. Atendiendo a su taxonomía; pertenecen al phylum *Molusco*, la clase *Gastropod*a y a la superfamilia *Conidae*. Destacan por sus conchas. Tienen un patrón característico en cada especie que las hace ideales para ser distinguidas por una red convolucional. Se detalla en la sección 1.5. Esto se debe a la capacidad de abstracción de patrones geométricos en las capas más altas de dichas redes. En las figuras 1, 2 y 3 se muestran algunos ejemplos de Conus. Se muestran con la resolución original. Como se puede apreciar, las fotografías se han entregado con distinto ancho y alto de píxeles. Esto se corregirá antes de pasárselos a la red. Se explica en el apartado 3.1.



Figuras 1,2 y 3. De izquierda a derecha se tienen muestras de *Aemulus* (proveniente de Angola, Este Atlántico), *Archon* (Pacífico Este) y *Ventricosus* (Mediterráneo).

El Museo Nacional de Ciencias Naturales ha aportado un fichero con cinco carpetas. Uno por cada localización geográfica. Se distinguen en Atlántico Este, Atlántico Oeste, Pacífico Este, Indo Pacífico y Mediterráneo. Para cada una de estas localizaciones encontramos a su vez nuevas carpetas; una por cada especie. Contienen varias fotografías con ejemplares de dicha especie.
La ruta de cada fotografía en nuestro terminal tendrá la siguiente estructura:

/Conus/Región_Geográfica/Número_de_foto-Nombre_Especie

Se tienen un total de 1988 muestras distintas. Se han distribuido aleatoriamente en 1762 muestras de entrenamiento y 196 de validación.
En las figuras 4 y 5 se muestran respectivamente los histogramas de entrenamiento y validación para saber cuantas muestras de cada especie hay. Es necesario para evitar que haya un gran desbalance de muestras (class imbalance) entre las distintas especies, ya que podría llevar a exponer a la red a muchas más muestras de una especie que de otra durante el entrenamiento. Esto podría provocar un sobreajuste (overfitting) de los parámetros de la red a unas de las especies mientras que otras serían distinguidas con mucha mayor dificultad. En el caso de las muestras de validación; como no alteran los parámetros de la red y se dispone de muchas menos muestras, el desbalance entre especies no tiene impacto en los resultados.

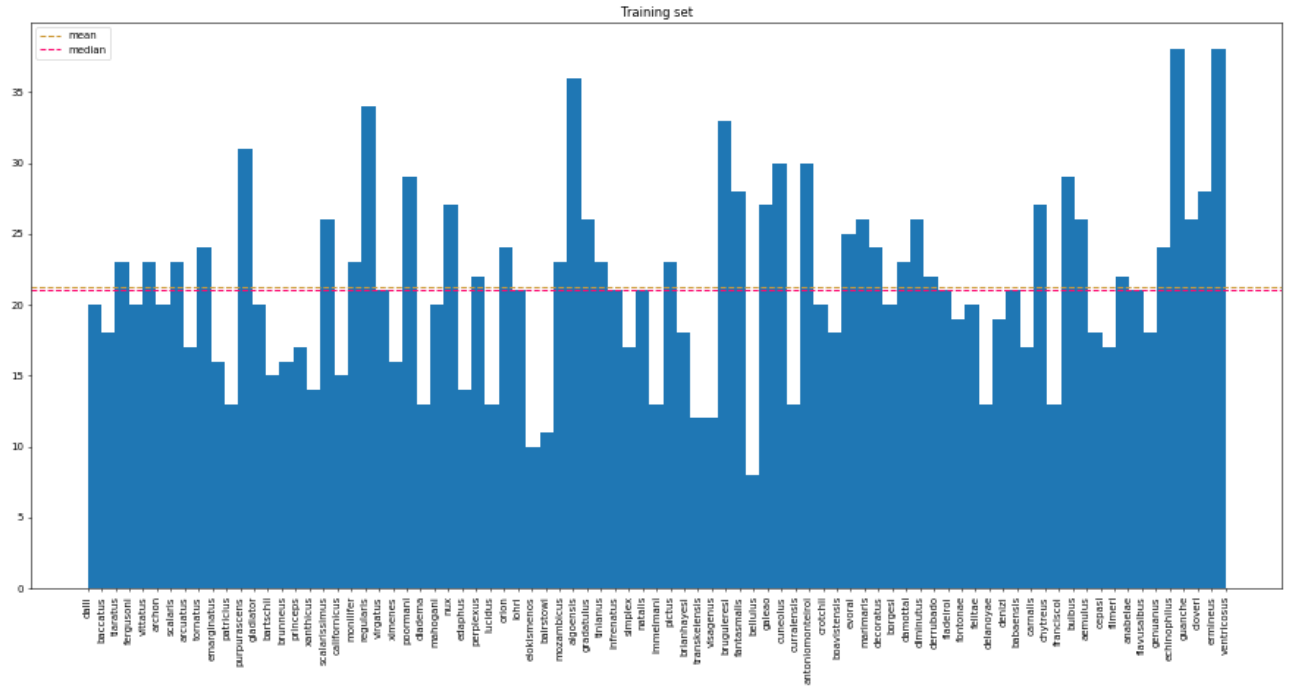

Figura 4. Histograma de especies para las muestras de entrenamiento.

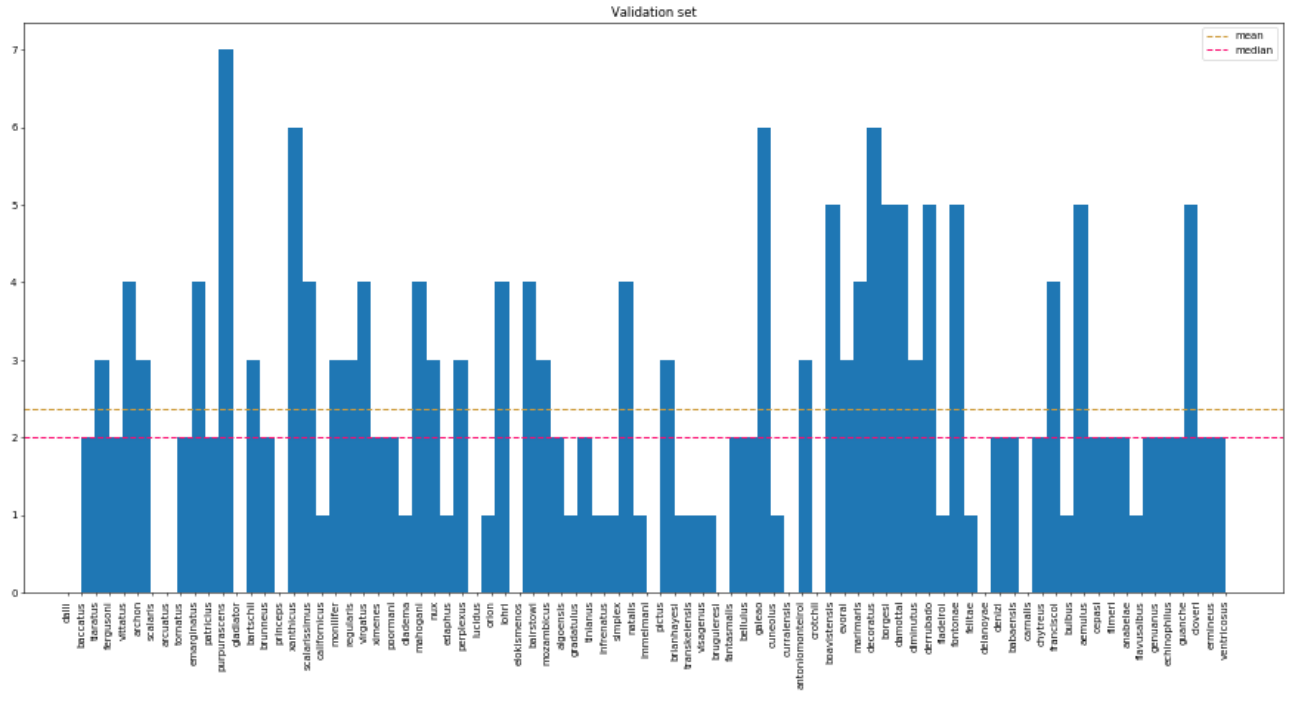

Figura 5. Histograma de distribución de especies para las muestras de validación

1.2 Estado del arte

Los procesos de aprendizaje profundo se basan en el producto de tensores de grandes dimensiones. También requiere de cálculo simbólico. Se detalla en las secciones 1.4 y posteriores. El coste computacional de dichos procesos es tal que técnicas como las empleadas en este trabajo no han podido ser implementadas hasta principios de la última década.

Hasta el comienzo de los 2010, la mayoría del cálculo mediante máquinas se realizaba en CPUs. La CPU o procesador de un ordenador está compuesta por núcleos. Generalmente se dispone de entre 2 y 8 núcleos. Un núcleo es la unidad de procesamiento básica de un ordenador. Puede realizar un único proceso a la vez. Para ello lo descompone en cuatro partes: lectura de los datos de entrada, decodificación, ejecución de las operaciones necesarias y escritura de los resultados. Se conoce como arquitectura de Von Neumann [2].

No a todos los núcleos les toma el mismo tiempo realizar las cuatro partes de dicho ciclo. Se caracterizan por el número de ciclos que pueden realizar por segundo. Hoy en día se encuentran entre los 2GHz y los 4 GHz.
Debido a que cada núcleo no puede realizar más de un proceso a la vez, se emplean los hilos. Son las unidades de instrucciones de procesamiento más pequeñas que el sistema operativo le da al procesador. Son comandos programados a bajo nivel que se encargan de repartir cada tarea que tiene que realizar el núcleo en porciones más pequeñas. De este modo, se van alternando dichas porciones, dando la sensación de que varias tareas se están realizando al mismo tiempo. Lo que está ocurriendo realmente es que se está optimizando el cómo se realizan dichas tareas para que tome el menor tiempo posible terminarlas.

Entre 1990 y 2010 la frecuencia de las CPU aumentó por un factor de aproximadamente 5000 [3].
Permitió realizar los primeros procesos de aprendizaje profundo. Sin embargo, seguía siendo insuficiente para modelos más extensos. Como se explicará en 1.4, el coste de dichos procesos se debe a la gran cantidad de operaciones a realizar y no de la complejidad de las mismas.
Incluso con ocho núcleos y 2 hilos, no se podían realizar simultáneamente más de 16 procesos [4].
Una red neuronal habitual puede tener entre varias decenas de miles hasta millones de coeficientes con sus respectivas operaciones. La arquitectura informática de la CPU no era adecuada para realizar modelos de aprendizaje profundo.
La arquitectura que resuelve este problema es la de las GPU (Graphical Processing Unit). Se conoce como Modelo Circulante. Mientras la CPU es la unidad de procesamiento general, la GPU se encarga de las operaciones en coma flotante. Estos procesos son menos complejos que los que resuelve la CPU, pero mucho más numerosos. Emplea para ello varias decenas de núcleos con menor capacidad de computación.
En lugar de seguir el proceso en cuatro partes, paraleliza todas estas operaciones en cientos de unidades de cálculo. Estas unidades de cálculo o tensor cores se comportan parecido a los hilos de un núcleo en una CPU, pero son mucho más numerosos.
Una gráfica moderna como la NVIDIA TITAN RTX empleada en este trabajo dispone de 72 de estos pequeños núcleos, para un total de 576 unidades de cálculo.

Normalmente el uso de GPU está asociado a procesar y mostrar las imágenes en la pantalla del ordenador. Sin embargo, es posible destinar esta capacidad de procesamiento para los cálculos deseados por el usuario. Para ello se emplea CUDA, un software basado en lenguaje a bajo nivel desarrollado por la compañía NVIDIA. Dispone de numerosas librerías que pueden ser usadas en lenguajes de programación a alto nivel como Python o R para, de modo sencillo, destinar a la gráfica todas las operaciones que normalmente serían resueltas en la CPU.

En la figura 6 se muestra una idea intuitiva de la diferencia entre una CPU y una gráfica.

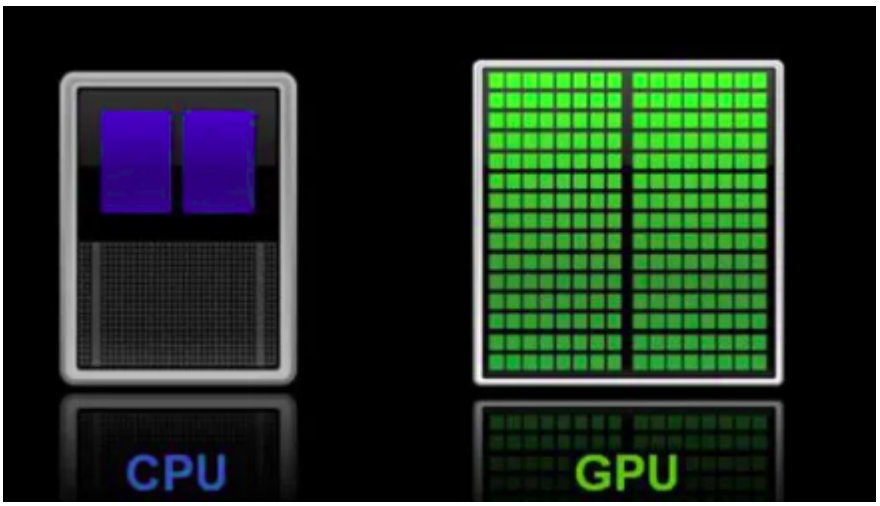

Figura 6. Diagrama con las diferencias entre la arquitectura Von Neumann de una CPU y el Modelo Circulante de una GPU. Aportada por la fuente [5] de bibliografía.

1.3 Aprendizaje automático

El machine learning o aprendizaje automático es una rama de la inteligencia artificial. Consiste en el desarrollo de software capaz de refinar por si mismo una tarea dados unos datos de entrada. Se emplea para procesar información y clasificarla o emplearla para predecir comportamientos futuros. Resuelve problemas de clasificación binaria, clasificación en múltiples categorías con una o más etiquetas y regresiones.

En función de su implementación se distinguen dos tipos de aprendizaje automático distinto: supervisado y no supervisado.

i) El aprendizaje no supervisado consiste en desarrollar software capaz de distinguir por sí mismo patrones y estructuras entre diversos datos de entrada aportados.

ii) El aprendizaje supervisado es el empleado en este proyecto. Consiste en implementar un programa al que se le entregan unos datos de entrada (input) y unos resultados asociados a ellos (outputs). Con dichas muestras y mediante distintos procesos; el programa “aprende” a asociar las entradas con las salidas. Se denomina entrenamiento. Después de entrenar podrá asociar salidas a nuevas entradas por sí mismo. La problemática consiste en mejorar la precisión con la que lo hace.

En nuestro caso, los datos de entrada son las imágenes de los ejemplares de gastropoda conidae y la región geográfica donde se encontraron. Las salidas son el nombre de la especie que aparece en cada foto.

En definitiva, se busca encontrar una serie de operaciones entre las predefinidas por el software tales que permitan transformar la información de entrada en datos con algún significado. Dicho conjunto de funciones se conoce el espacio de hipótesis de nuestro problema.

1.4 Aprendizaje profundo

Dentro de los modelos de aprendizaje automático supervisado, el objeto de este trabajo es el aprendizaje profundo o deep learning. En la figura 7 se presenta un esquema de su funcionamiento.

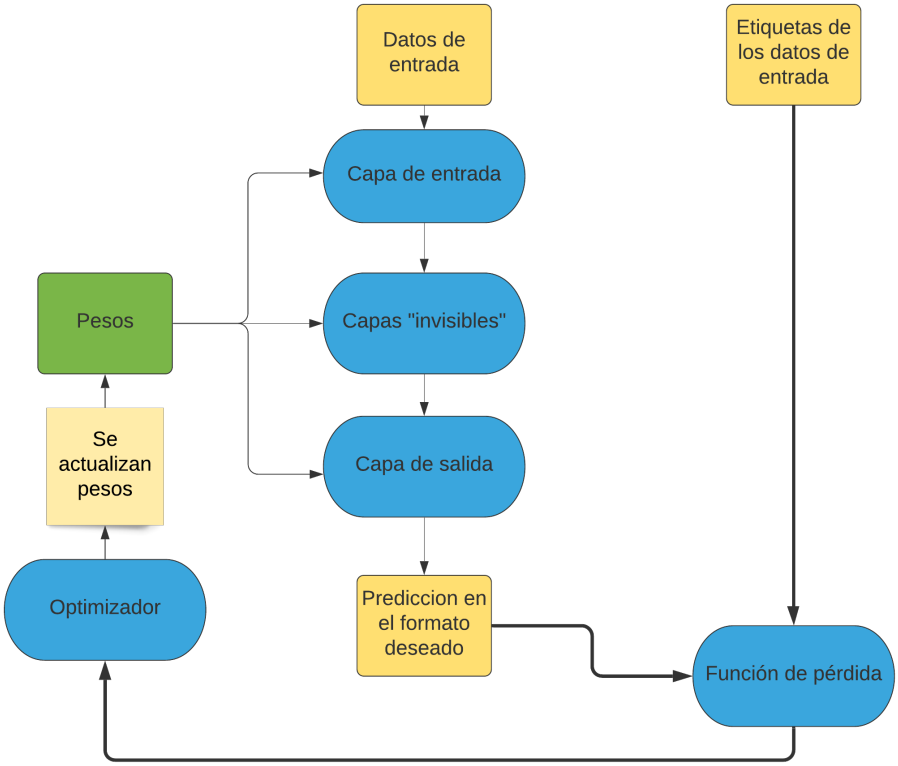

Figura 7. Diagrama de flujo de los distintos procesos realizados durante el aprendizaje profundo.

El deep learning consiste en aplicar el método de descenso del gradiente. Antes de pormenorizar su funcionamiento, es necesario tener una idea general del mismo.
Previo a empezar, los datos de entrada tienen que procesarse para expresarse como un tensor.
Una vez preparados, se introducen en el modelo o "red neuronal". El modelo se compone de distintas "capas". Cada capa va tomando el tensor que sale de la anterior y le aplica tres operaciones. La primera es un producto con otro tensor W^i característico de dicha capa. La segunda es sumarle otro tensor B^i, también distinto para cada capa. Por último, se le aplica una función de activación f^i.
La función de activación de la última capa retorna la predicción de qué etiqueta le corresponde a la muestra de entrada. Con esto, se computa la función de pérdida L. Es proporcional a la diferencia entre la predicción y la etiqueta real. Además, es una función dependiente de los coeficientes W^i_j de cada capa. Es decir,

$$L = L(W_{1,\ldots\ldots,}W_n) \qquad (1)$$

Después, se calcula el gradiente de L y se encuentra su mínimo. Así es posible saber cómo variar ligeramente los coeficientes para ir minimizando la pérdida:

$$W^i_j = W^i_j \pm alpha\frac{dL}{dW^i_j} \qquad (2)$$

Donde $alpha$ es el "learning rate". Es un factor que el optimizador va modificando para evitar oscilar en torno a un mínimo. Se detalla más adelante en esta sección.

Tras aplicar (2), se toma un nuevo batch y se vuelve a repetir todo el proceso para refinar el modelo. Cada repetición recibe el nombre de época. Así, siguiendo esta sucesión de operaciones tensoriales, se consigue "mapear" las muestras a su etiqueta correcta.

Ahora vamos a ver cada uno de los pasos descritos con un poco más de detalle:

i) Los datos de entrada *X* se preprocesan para expresarse en forma de tensor. En el caso de este trabajo, cada muestra es una fotografía a color y se convierte en un tensor de tamaño (3,224,224) donde 224 es el ancho y alto de píxeles. El 3 se debe a que una fotografía a color se descompone en 3 fotografías RGB (una para color primario).
Generalmente, el tensor X que se le pasa al modelo contiene la información de varios datos de entrada en bloque o "batch". El número de muestras que lleva cada bloque se conoce como *batch size*. En nuestro caso, se le pasan 16 fotografías a la vez, por lo que el tensor de entrada al modelo tiene la forma $(16,3,224,224)$.
Para prevenir que los coeficientes X_i del tensor X puedan hacerse demasiado grandes durante el proceso, se normalizan. De modo que:

$$X_{\text{normalizado}} = \frac{X - \overline{X}}{\sigma_X} \qquad (3)$$

Donde X es el tensor original. $\overline{X}$ es el valor medio de los coeficientes X_i y σ_X su desviación estándar.

Por otro lado, la etiqueta real que tiene cada muestra se procesa también en forma de tensor. Se suele preparar en formato "one-hot-encoding". Consiste en tener una fila para cada muestra de entrada y una columna para cada una de las etiquetas posibles. De este modo, todos los elementos de cada fila serán 0 salvo el que esté en la columna asignada a su especie real. Ese tomará 1 como valor. En nuestro caso, para cada batch se genera una matriz de la forma (16, 83) pues el *batch size* es 16 y tenemos 83 especies distintas a las que pueden pertenecer las muestras.

ii) El batch de datos de entrada X^0 es insertado en la primera capa de la red. Cada capa i-ésima opera sobre el tensor de entrada X^{i-1} para obtener unos datos de salida X^i de la siguiente manera:

$$X^i = f^i(W^i \cdot X^{i-1} + B^i) \qquad (4)$$

Los tensores W^i y B^i son característicos de la capa i-esima de la red. Se ha notado su producto interno con "·". Los coeficientes de W^i se modifican en cada capa de la red hasta que la salida obtenida sea próxima a la etiqueta esperada. Se ha explicado en la ecuación (2).

Es posible escoger el tamaño y forma de los tensores de cada capa. Así, el tensor de salida tendrá mayor o menor cantidad de componentes. La cantidad de coeficientes $W^i{}_j$ de los tensores W^i tiene un papel determinante en la precisión o accuracy del modelo. Si se tiene un exceso de coeficientes, se corre el riesgo de que el modelo se ajuste demasiado a los datos de entrada. Si se tienen pocos coeficientes, puede que no sean suficientes para ajustar la red apropiadamente a los batch de entrada y que las predicciones realizadas tengan una precisión inferior a la deseada. Esto se comenta en la sección 1.7.

f^{i} es la función de activación. Junto con W^{i} y B^{i} caracteriza a la capa. Esta función debe tener una serie de propiedades mínimas, tales como ser diferenciable y su derivada suave. Ejemplo de ello son las funciones de clase C^{1}. No existe una regla general para escoger dicha función. Existen numerosas publicaciones que debaten de forma experimental cuál es mejor en cada situación [6]. En este caso se utiliza la denominada función de activación *ReLU* (Rectified Linear Unit) f_{relu}. Toma el valor máximo entre 0 y $x^{i}{}_{j}$ para cada coeficiente $x^{i}{}_{j}$ del tensor X^{i} al que se aplique.

$$f_{relu}(x^{i}{}_{j}) = \begin{matrix} x^{i}{}_{j} & \text{si} \quad x^{i}{}_{j} > 0 \\ 0 & \text{en otro caso} \end{matrix} \qquad (5)$$

iii) Tras pasar por las capas intermedias o "invisibles", el tensor llega a la última capa n-ésima o de salida. Esta tiene una peculiaridad respecto a las anteriores. Su función de activación es distinta ya que su salida debe tener el formato de la etiqueta deseada.
En nuestro caso, el tensor, para cada muestra del batch, tendrá 83 componentes antes de aplicar la última activación. Como nuestro modelo resuelve un problema de clasificación entre múltiples etiquetas, se aplica la activación más habitual en estos casos. Se llama *Softmax*. Toma como entrada un vector X^{i} con *n* componentes reales y retorna una distribución discreta normalizada con *n* probabilidades. Como fórmula general para un vector de entrada X^{i} de *n* componentes de la forma $x^{i}{}_{j}$:

$$f_{softmax}(x^{i}{}_{j}) = \frac{e^{x^{i}{}_{j}}}{\sum_{k=1}^{k=n} e^{x^{i}{}_{k}}} \text{ para j} = 1....n \qquad (6)$$

Como resultado tras aplicar dicha activación se tendrá una distribución de probabilidad discreta normalizada a la unidad para cada una de las muestras del batch. Formalmente será un vector cuya componente n-ésima dan la probabilidad de que la imagen de entrada perteneza a la categoría *n* de entre todas las posibles.
En nuestro caso *n* vale 83; las 83 especies distintas de Conus. De modo que tendremos como salida una matriz con 16 filas y 83 columnas. Cada fila puede interpretarse como un vector de la forma:

$$X^{salida} = (x^{salida}{}_{1}, x^{salida}{}_{2}, \ldots\ldots, x^{salida}{}_{83}) \qquad (7)$$

Cada componente nos da la probabilidad de que dicha muestra pertenezca a la especie que indica esa columna.

iv) Tras realizar todo este proceso, se tiene una distribución discreta de probabilidades para cada muestra del batch. Está normalizada a la unidad. En nuestro caso, tenemos 83 posibles especies, por lo que dicha distribución para cada batch viene dada por un vector como (7). Entre todos forman una matriz de tamaño (16, 83). El coeficiente más alto $x^{salida}{}_{Max}$ en cada distribución se corresponde con la especie que está prediciendo la red para dicha muestra.

Con esto, ya podemos calcular la función de pérdida L. Se ha empleado la función *Categorical Cross-Entropy Loss* L_{CE}. Se habla sobre ella en la referencia [7] de bibliografía. Viene dada por la siguiente fórmula para cada muestra del batch:

$$L_{CE} = -\log\left(\frac{e^{x^{salida}_{Max}}}{\sum_{k=1}^{k=n} e^{x^{salida}_{k}}}\right) \qquad (8)$$

Notemos que x^{salida}_{k} es la probabilidad discreta de que dicha muestra pertenezca a la especie k-ésima. También recordamos que *n* es el número de etiquetas y que en nuestro caso son las 83 especies distintas.

Promediando las funciones de pérdida de todas las muestras del batch se obtiene la pérdida $\overline{L_{CE}}$ del modelo en cada época del entrenamiento.

v) Acabamos de calcular numéricamente el valor de la pérdida. Como se ha explicado en la ecuación(1), analíticamente la pérdida es función de los tensores de la red W^{i}. Se calcula simbólicamente el gradiente de la pérdida L_{CE} y se iguala a 0 para encontrar sus mínimos. Se varían ligeramente como en la ecuación (2) para minimizar dicha pérdida. Así, el valor dado por (8) será menor en cada iteración. Esto implicará que cada vez se están identificando correctamente más muestras. A nivel de software, en nuestro programa esto lo realiza la librería de redes neuronales Keras [8].

Existe el riesgo de que el la variación en cada coeficiente $\frac{dL}{dW^{i}_{j}}$ de la red sea demasiado grande. Estaríamos entonces oscilando en torno a un mínimo indefinidamente. Llegaría un punto en el que la pérdida no se reduciría en cada época. Para evitar esto se usa el optimizador. Se ha usado *Adam*. Se trata de un coeficiente $alpha$ que multiplica a $\frac{dL}{dW^{i}_{j}}$ y que se va reduciendo cada época que la función de pérdida se reduce.

Un comando habitual para crear una capa en nuestra red o modelo tiene la siguiente forma en Python:

```
model.add(layers.Dense(4, activation = 'relu'))
```

Donde se está indicando al programa el tipo de activación de la capa y el tamaño de la primera dimensión del tensor que retorna.

1.5 Redes convolucionales

Dentro del aprendizaje profundo, distinguir imágenes o patrones como en el caso de este trabajo se conoce como visión artifical o computer vision. Para esta tarea se combinan las capas antes descritas con otras que han aportado mejoras en la precisión o accuracy significativas durante los últimos años. Se llaman redes convolucionales.

Una red convolucional es similar a una red habitual o densa como las de la sección 1.3.
Sin embargo, antes de aplicar la ecuación (4) al tensor de entrada, le pasa un “filtro”.
Un ejemplo de su declaración en Python (con Keras) es el siguiente:

```
model.add(layers.Conv2D(32, (3,3), activation='relu'))
```

Podemos notar que es similar a la creación de una capa densa. Tiene su activación *ReLU*, y 32 como tamaño de la primera componente del tensor de salida.
Por tanto, si se le aplica a los datos de entrada de una imagen a color de 224x224 píxeles como las de este trabajo, la salida sería un tensor con forma (32, 224,224). Es decir, la salida serían 32 matrices de tamaño (224,224). Cada una de ellas tiene guardada información relevante sobre la imágen original. Es como si se le hubiera pasado un filtro que reconoce algún aspecto relevante de la foto y lo almacenara remarcándolo.
Durante el aprendizaje previo al TFG, se han estudiado estas redes y en la figura 8 se muestran ejemplos de lo que le hacen estos filtros a las imágenes de un gato usando una red entrenada para distinguir entre gatos y perros.

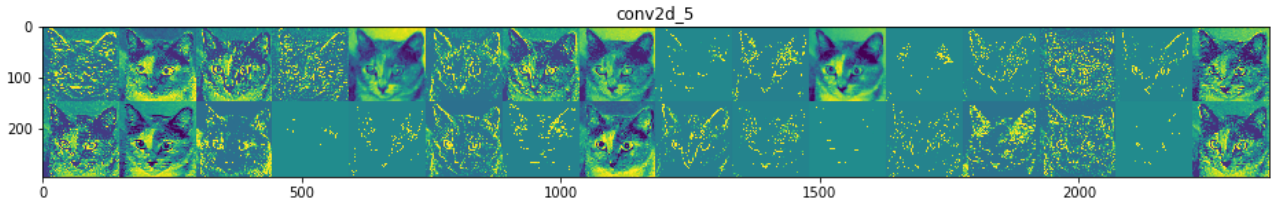

Figura 8. Representación del tensor de salida tras aplicarle una capa convolucional a la imagen de un gato.

Como se observa, unos filtros resaltan las orejas, otros los ojos, bigotes y otro tipo de información. Cada una de estas 32 matrices de salida codifica información relevante sobre la figura que permitirá a la red decidir qué clase de etiqueta hay que poner (si es un perro o un gato).
Las redes convolucionales, en lugar de aprender un patrón general en los datos de entrada como las densas, aprenden formas aisladas. Luego son capaces de reconocerlas en cualquier parte de la figura. Experimentalmente se observa que, si se aplican varias capas convolucionales, las que estan más arriba y se aplican primero, aprenden caracterísitcas de bajo nivel que son generales para cualquier problema de clasificación de imágenes. Según avanzamos, las últimas capas aprenden características de alto nivel más representativas del problema específico que estamos tratando. Esta idea se resume en la figura 9 aportada por la fuente [3] de bibliografía.

Todo esto motiva el concepto transfer learning. Es posible reutilizar las primeras capas, con sus respectivos coeficientes, de una red previamente entrenada para otro problema de clasificación de imágenes. Puede ser muy beneficioso cuando se disponen de pocas muestras para el modelo que se está abordando. Al contener las primeras capas convolucionales características generales que se aplican a muy diversos objetos, podemos “reciclar” las capas que se han entrenado en otra situación con muchas más muestras. Además, se libera a la GPU de mucho trabajo y memoria que puede optimizar en entrenar más ciclos las redes bajas que atienden a aspectos más específicos del probema a resolver. Esto se ha realizado en este trabajo con el modelo Xception [10].

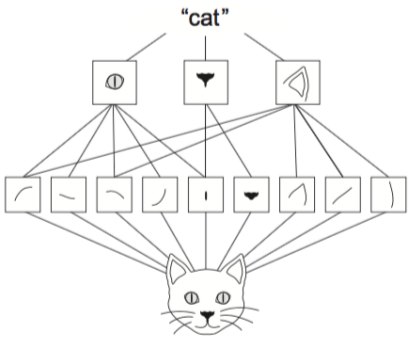

Figura 9. Esquema de las abstracciones de una red convolucional. Aprende jerarquías espaciales.

Así, se libera a la GPU de mucho trabajo y memoria que se puede emplear en entrenar más ciclos las redes bajas que atienden a aspectos más específicos del problema a resolver.

Dentro de las redes convolucionales distinguimos dos tipos de capas: Conv2D y MaxPooling2D:

i) Conv2D. Tenemos un ejemplo de su declaración en Python en la página previa. Notamos que a diferencia de las redes densas, ahora hay una variable que tiene el valor (3,3). Esto se conoce como rejilla (grid). Se dedica a recorrer todos los coeficientes de cada matriz que compone el tensor de entrada. Los sustituye por el promedio de los coeficientes de la matriz adjunta 3x3 centrada en el coeficiente original. De este modo, el tamaño de las matrices del tensor de salida será menor que el original. Se debe a que no se puede centrar una rejilla en los bordes. Al promediar varios píxeles próximos, se eliminan aspectos más particulares de la imagen y se observan rasgos más generales. Después de esto, se aplica la fórmula (4) con su respectiva matriz de coeficientes y función de activación.

ii) Las MaxPooling2D son capas que carecen de matrices de coeficientes W o activación f . Simplemente pasan una rejilla 2x2 sobre el tensor de entrada de forma alterna y toman el valor máximo encontrado en cada rejilla. De este modo, las matrices que componen el tensor ven reducido su tamaño (*m*,*n*) a $(\frac{m}{2},\frac{n}{2})$. Sirve para disminuir rápidamente el tamaño de las muestras y empaquetar en la mitad de espacio la información original. Así, cada coeficiente contiene "más información" que los que había en el tensor de entrada . Facilita el trabajo de la red para distinguir figuras. Se puede declarar en python con un comando como este:

```
model.add(layers.MaxPooling2D((2,2))
```

Tras aplicarse las distintas capas convolucionales al tensor de entrada, se pasa la salida a una red densa habitual como las descritas en 1.4. Esto se realiza mediante una capa que convierte la lista de matrices para cada filtro en un vector. Hay dos formas de hacerlo:

i) *Flatten*. Este método consiste en introducir directamente todos los coeficientes del tensor de salida en un único vector. Se disponen en fila y se pasan a las capas de la red densa. Así, si por ejemplo el tensor de salida tras aplicar las capas convolucionales del modelo tiene la forma (4,32,32) el vector resultante tendrá 4x32x32=4096 elementos. Los desarrolladores de la propia librería Keras indican que realizar *Flatten* aumenta significativamente la precisión de forma experimental cuando se está trabajando con muchas muestras de entrada [9].

ii) *Max Average Pooling*. Se calcula el promedio de los coeficientes de cada una de las matrices que forman el tensor de salida. Retorna un vector cuyas componentes son estos promedios. Funciona bien cuando las muestras no son grandes o no hay suficiente memoria RAM en la GPU para almacenar todos los parámetros que provoca hacer *Flatten*.

1.6 Datos de entrenamiento, validación y matrices de confusión

Cuando la red está entrenando, es necesario tomar algún tipo de referencia del progreso. La precisión o accuracy en las predicciones de los datos de entrenamiento es importante. Sin embargo, no nos da idea de la generalidad con la que se está entrenando la red.
Por esto, cuando se entrena un modelo, es conveniente separar los datos en dos conjuntos:

i) Los datos de entrenamiento son los que se emplean para entrenar la red con el modelo de descenso del gradiente descrito en la sección 1.4.

ii) Los datos de validación son los que se usan para evaluar el modelo al final de cada época. Se predicen sus etiquetas y se comparan con las reales. No alteran los coeficientes W^{i}_{j} de la red pero permiten tener una idea de cómo va progresando su capacidad de predicción.

En este trabajo se ha destina el 10% de los datos totales a validación. El 90% restante se emplea en entrenar la red. Es necesario que se decida aleatoriamente qué muestras se usan para validación y qué otras para entrenamiento.
Se debe evitar que todas las muestras de validación pertenezcan a las mismas clases. Así, no se tiene el riesgo de que la red solamente haya aprendido a distinguir bien unas pocas clases y que resulte que esas son las que se están comprobando.

Las matrices de confusión se construyen con un fin similar. Evalúan el modelo final. Consisten en una tabla de doble entrada. Las filas son las categorías reales de cada muestra. Las columnas son las categorías predichas con la red una vez se ha entrenado. Para nuestro caso se muestran en la sección de análisis 4.
Cada vez que se predice una muestra de conus del conjunto de validación, se va a la fila de la tabla que se corresponde con la especie a la que pertenece realmente dicha muestra. Se busca la columna que se ha predicho y se marca. Esto se realiza para todo el set de validación. En lugar de marcas muchas veces se usa un mapa de colores. Si todas las muestras se han predicho correctamente, todas las casillas que se han marcado formarán una diagonal.
Cuanto más diagonal sea la matriz de confusión, mejor será el modelo.

El *top_k* consiste en calcular para las muestras de validación qué porcentaje de las ocasiones la especie real que aparece en la imágen se ha encontrado entre las k especies con más probabilidades según la red.

1.7 Aumentación y sobreajuste

Cuando se tienen pocas muestras de entrada para entrenar la red, existe el riesgo de que se produzca sobreajuste u overfitting de los parámetros. La red entrena tantas veces con pocas muestras que acaba “mapeando” las entradas con las salidas sin que esos coeficientes $W^{i}{}_{j}$ den cuenta de las generalidades de la red. Se tiene un modelo que predice con casi un 100% de precisión las muestras de entrenamiento. Sin embargo, si se le expone a nuevas imágenes con las que no ha entrenado, las predicciones probablemente sean erróneas. Cuando esto ocurre podemos notar que en cada época la precisión de las muestras de validación deja de mejorar. Se profundiza en la sección 4 de resultados y análisis. Un ejemplo de ello se muestra en la figura 10.

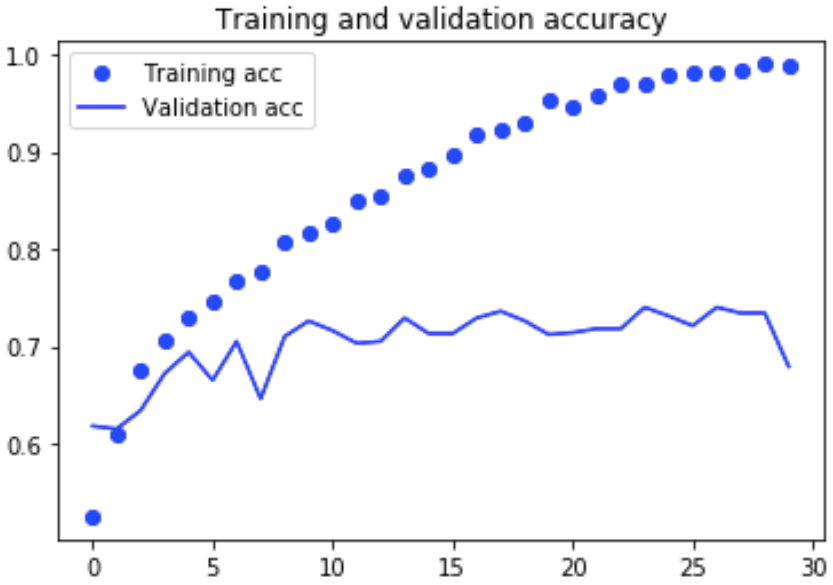

Figura 10. Aunque la precisión en las predicciones con las muestras de entrenamiento mejora época a época, las de validación no progresan. La red está aprendiendo características particulares de las muestras de entrenamiento y no aspectos generales que le permitan diferenciar nuevas muestras.

Para evitar esto existen técnicas de aumentación. Consiste en editar las imágenes de entrada para generar otras que, aunque no son nuevas ya que son modificaciones de las originales, son distintas. Como realmente no se está aportando información nueva, la aumentación tiene sus limitaciones como herramienta para combatir el sobreajuste. Siempre es preferible disponer de una mayor cantidad de muestras originales. Aunque esto no siempre es posible.

2. Herramientas y materiales

El proyecto se ha desarrollado dentro de un Docker de clasificación de imágenes genéricas creado por el grupo de e-Ciencia y computación avanzada del Instituto de Física de Cantabria.
Se detalla en la sección 2.2. Contiene el código en Python para entrenar una red que distinga imágenes de objetos cotidianos y las etiquete. Es necesario pasarle imágenes de entrenamiento junto con la clase (especie) a la que pertenecen. También es necesario pasarle muestras de validación. Sirven para calcular la precisión real del modelo tras cada época de entrenamiento. Incluye también una red ya preentrenada para dichas imágenes genéricas. Dispone de una API funcional para que el usuario pueda predecir sus muestras. Todo esto se desarrolla en las subsección 2.2.

Todo el código original, así como el que se ha debido modificar y construir para este trabajo se ha desarrollado en Python 3. El motivo es que Python dispone de la librería Keras. Esto se explica en la sección 2.1.

La GPU empleada en este proyecto es una NVIDIA TITAN RTX.

2.1 Python, Keras y Tensorflow

La mayor parte del esfuerzo realizado en este trabajo se concentra en entender la problemática concreta y desarrollar el código óptimo de la red. También el del preprocesado de las muestras y el análisis de la red tras entrenar.
Se ha empleado Python en su versión 3.5.
Ha sido necesario aprender a utilizar dicho lenguaje. Se han realizado diversas prácticas de introducción a Python como paso previo a la realización de este Trabajo Fin de Grado. Como el código de las mismas excedería el máximo de páginas permitido, se incluye un enlace a mi repositorio de GitHub en la referencia [11] de bibliografía. En él es posible ver y descargar dichos ejercicios.
Se debe a dos motivos:

i) Python es un lenguaje de programación orientada a objetos. Es posible asociar una serie de variables y funciones a realizar con ellas. Esto se conoce como clase. Es una abstracción. Cuando a estas variables se les da un valor concreto, estamos generando un objeto de dicha clase. El beneficio principal es que dicho objeto dispondrá de todas las funciones asociadas a esa clase. Una clase puede ser “hija” de otra. Basta con indicar entre paréntesis el nombre de la clase padre tras nombrar la que estamos desarrollando. La “hija” pasará a disponer de todas las funciones y variables de la padre. Este mecanismo de herencia es muy beneficioso cuando los objetos son, por ejemplo, capas de una red.

ii) Debido a lo expuesto en i), la empresa Google escogió Python como lenguaje para su librería de redes neuronales Keras [8]. Se trata de la librería más popular en el ámbito del deep learning. Las capas del modelo son objetos de esta librería y se pueden importar a nuestro código. El modelo en sí también es un objeto. Las activaciones ya están creadas en Keras y pueden declararse como atributos de la capa que estemos usando. Las operaciones como (4) de la introducción ya son funciones pertenecientes a la clase “capa”.

Keras no realiza directamente las operaciones como (4). En su lugar, emplea de forma auxiliar otra biblioteca llamada Tensorflow. Por tanto, Tensorflow actúa como “backend” de Keras. Tensorflow es una librería de C++, pero cuenta con un interfaz de Python. Se encarga de generar “sesiones”. Una sesión es un entorno de ejecución de todas las operaciones que se le pasan a Tensorflow. Reparte los cálculos entre la CPU y la GPU en función de cuál se adecúa más a la tarea. Otimiza el tiempo y precisión de los cálculos. Si se dispone de varias GPU o CPU también

es capaz de distribuirlo entre todas. Esta arquitectura en paralelo permite entrenar varias partes del modelo al mismo tiempo, o el mismo modelo con distintos datos de entrada.
Tensorflow dirige estos cálculos mediante CUDA como lenguaje a bajo nivel en caso de usar GPU. Para CPU emplea BLAS.
Python, Keras y Tensorflow vienen instalados en el Docker para facilitar su portabilidad entre varios usuarios. CUDA y BLAS deben instalarse en la máquina en la que se esté corriendo pues son los controladores de dicho Hardware. En la figura 11 se muestra un esquema de todo el proceso.

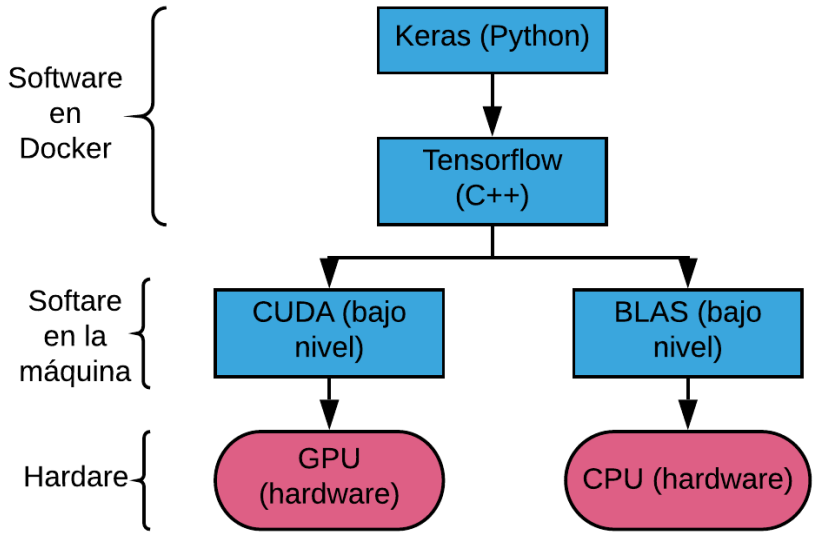

Figura 11. Diagrama que muestra las dependencias entre el distinto software y hardware necesario para entrenar una red neuronal.

2.2 Docker

El Docker o “contenedor” es un paquete de virtualización a nivel de sistema operativo en el que se almacena y ejecuta el software necesario para poder desarrollar el código de este trabajo. Es similar al uso de una máquina virtual. Puede ser ejecutado desde el terminal de cualquier ordenador que disponga de Windows o distribuciones Linux como Ubuntu. Se descarga de forma similar a una imagen o ISO de un sistema operativo habitual, pero se ejecuta como si fuera una aplicación normal desde nuestro terminal. Sus imágenes están escritas en su propio lenguaje, llamado “lenguaje Docker”. Contienen todo lo necesario para instalar este sistema operativo junto con todos los programas que desee su creador, así como contener ficheros, código en Python y cualquier otro software usual. De este modo, el usuario puede formar parte de un proyecto sin necesidad de tener que instalar en su máquina todas las aplicaciones y ficheros necesarios.

Es posible guardar y ejecutar instancias del Docker. Sería similar a tener varias copias de guardado o recuperación de nuestro sistema operativo con distintas fechas y contenido y ejecutar la que nos interese. Estas instancias se pueden compartir para que otro usuario se encuentre con todo exactamente igual a como lo tenía quien ha desarrollado lo que contiene dicho Docker. Facilita la portabilidad y los proyectos de software en grupo.

A diferencia de una máquina virtual, un docker es “transparente” al sistema operativo desde el que se está ejecutando. Podemos intercambiar contenido desde nuestro sistema original al docker y vicieversa con comandos simples desde el terminal de cualquiera de ellos. También es posible ejecutar aplicaciones y enviarlas como entrada a puertos de nuestro ordenador. De este modo, si tenemos una API (aplicación basada en html), podemos acceder desde cualquier navegador a dicho puerto y ejecutar la aplicación que hayamos creado cómodamente.

En el caso de este proyecto, se ha usado el docker “deep-oc-image-classification-tf” desarrollado por el departamento de computación avanzada y e-Ciencia del Instituto de Física de Cantabria. Es posible descargarlo y ver una pequeña descripción de los autores desde la referencia [12] de bibliografía. Contiene instalado tanto Python3, como Keras y Tensorflow. Así mismo tiene programado en Python el código necesario para entrenar una red que distinta imágenes genéricas. Este código se ha modificado para entrenar los Conus dando al usuario no solamente la opción de predecir la especie con imágenes sino también especificar la localización geográfica donde se encontró dicho espécimen y así mejorar dicha predicción. Esto se desarrolla en el método experimental.

Este Docker contiene además los parámetros de una red entrenada previamente para distinguir qué objetos cotidianos salen en las fotografías que le pasemos. Dispone además de una API funcional que permite ejecutarse desde un puerto de nuestro ordenador. En ella sacaremos por pantalla un interfaz en el que el usuario dispone de distintos botones para entrenar la red sin tener que tocar el código directamente, subir una foto y predecir de qué objeto se trata y modificar la red convolucional preentrenada que se va a usar para dichas predicciones.

De esta forma se tiene un entorno más amigable que permite a usuarios que no están familiarizados con el uso de Dockers o la programación en Python identificar sus muestras como si lo estuvieran haciendo en una página web.

En la figura 12 se muestra una captura del docker ejecutado desde el terminal. En la figuras 13,14 y 15 se muestra una captura de la aplicación funcional ejecutandose en un puerto del ordenador como si de una página web se tratase.

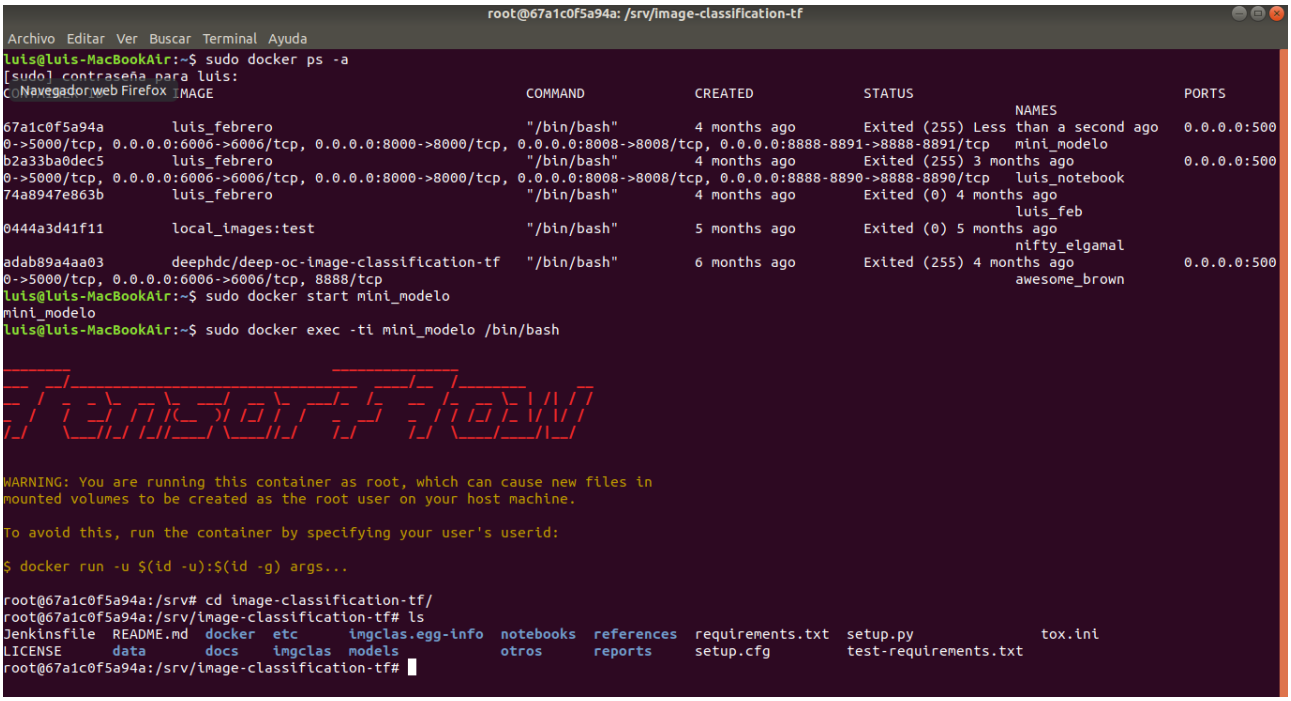

Figura 12. Ejecución del Docker desde un terminal de Ubuntu. En el momento en el que se comienza a ejecutar el docker, el usuario pasa de ser luis@luis_MacBookAir a root@67a1c0f5a94a. Hemos pasado al interior del Docker y estamos en su terminal.

Figura 13. Ejecución de la API desde el puerto 5000 de nuestra máquina. Se ven distintas pestañas que contienen opciones para realizar diversos procesos sin tener que modificar el código de Python en el interior del Docker. Destaca “predict”, el cual podemos desplegar y arrastrar un fichero para que prediga qué especie es la de la imagen.

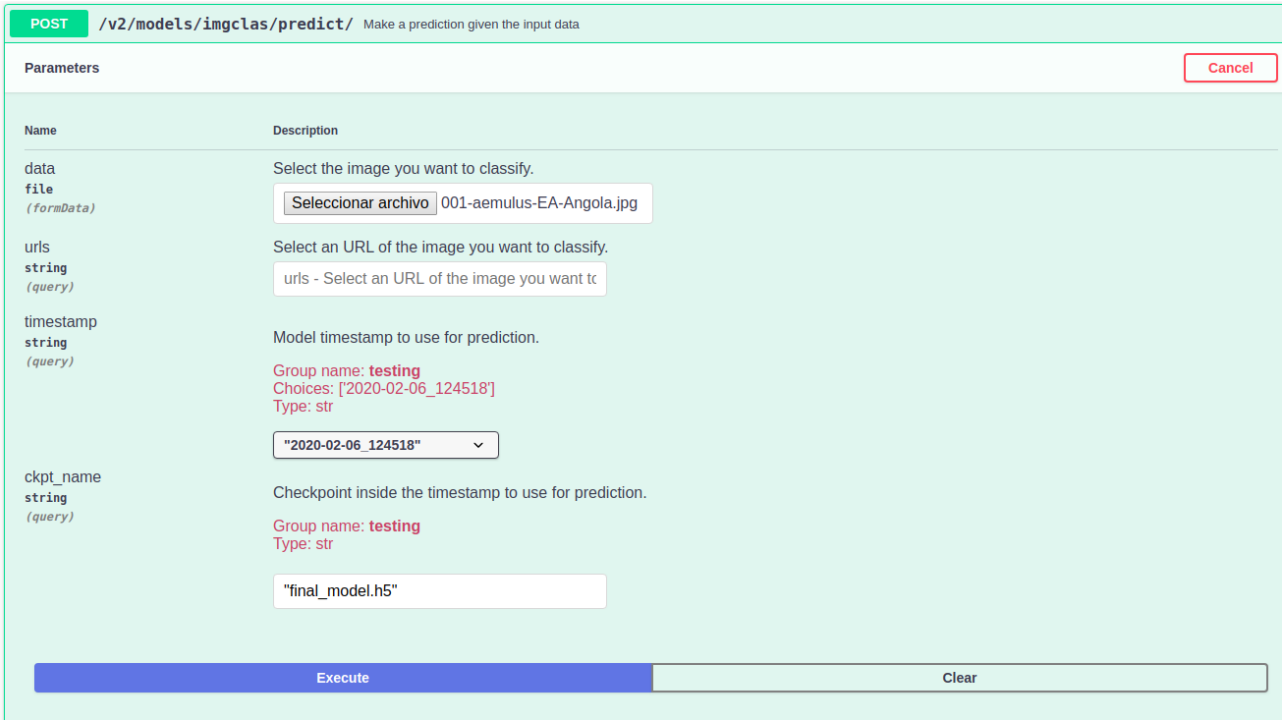

Figura 14. Se ha desplegado la opción “predict” Permite escoger el modelo (con sus respectivos coeficientes tras entrenar) de la red para predecir.

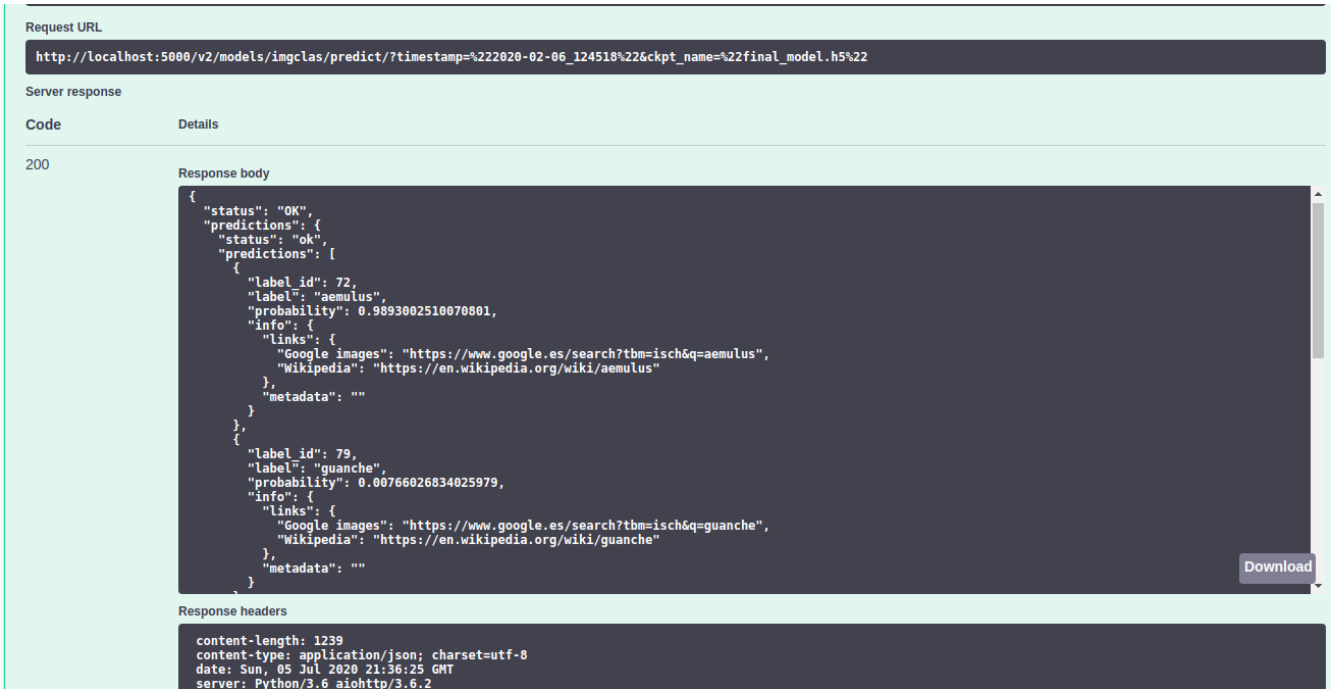

Figura 15. Resultado de la predicción en la API. A indicado que hay ~ 98.9 % de posibilidades de que la fotografía sea un Conus de la especie Aemulus. La fotografía que se le ha pasado pertenece a esta especie. Ha predicho satisfactoriamente.

3. Método experimental

Todo el código implementado en este trabajo fin de grado se encuentra subido a la referencia [13] de bibliografía.

Se han desarrollado varios scripts en Python para preparar las muestras y dividirlas en entradas de entrenamiento y validación del modelo. Las de entrenamiento son las que se emplean para calcular los parámetros de la red siguiendo el gradiente inverso explicado en la introducción. Las de validación sirven para predecir al acabar cada ciclo o época del entrenamiento. De este modo se ve ciclo tras ciclo del gradiente inverso cuánto va mejorando la precisión de la red para distinguir especies de conus con imágenes a las que no se ha ajustado directamente. Todo este código se explica en la sección 3.1. El código se encuentra en el fichero “preprocesar_muestras”.

Se ha entrenado la red para que distinga Conus respetando el modelo original del Docker “image-classification-tf” desarrollado por el grupo de e-Ciencia del IFCA. Se ha utilizado como red convolucional de base la red Xception. Se trata de un modelo de 36 capas convolucionales entrenado por Google para distinguir entre más de 1000 objetos distintos de la base de datos ImageNet [14]. Al ser una red convolucional que distingue patrones asociados a objetos cotidianos, ya tiene unos coeficientes próximos a los que queremos para distinguir Conus pues las abstracciones, patrones y formas que ha aprendido son similares. Así, podemos evitar tener que entrenar todas esas capas, lo cual cuesta mucho trabajo para la GPU y puede exceder su memoria. Bastará con que la red entrene en las capas inferiores para ajustarse a las imágenes de las distintas especies de Conus con más precisión.

Después se ha modificado el código para que la red admita localizaciones. También se han tenido que preparar las muestras para pasarle a la red dicho dato. Todo el código asociado a entrenar la red, preprocesar las imágenes para que tengan el mismo formato, aplicarles aumentación, y guardar los resultados finales se encuentra en la carpeta “Imgclas_lo”. Se explica con detalle en la sección 3.2.

Finalmente, para comparar la calidad de los distintos modelos se ha programado un código al que se pasan imágenes de entrada y la red con los coeficientes en la época que queremos analizar. Con ello el programa genera la matriz de confusión del modelo e indica el *top_k*. El código se encuentra en el fichero “calcular_matrices_confusion”.

En la figura 16 se muestra un diagrama que explica el flujo de los datos desde su procesado hasta la obtención de la matriz de confusión y pasando por el entrenamiento de la red.

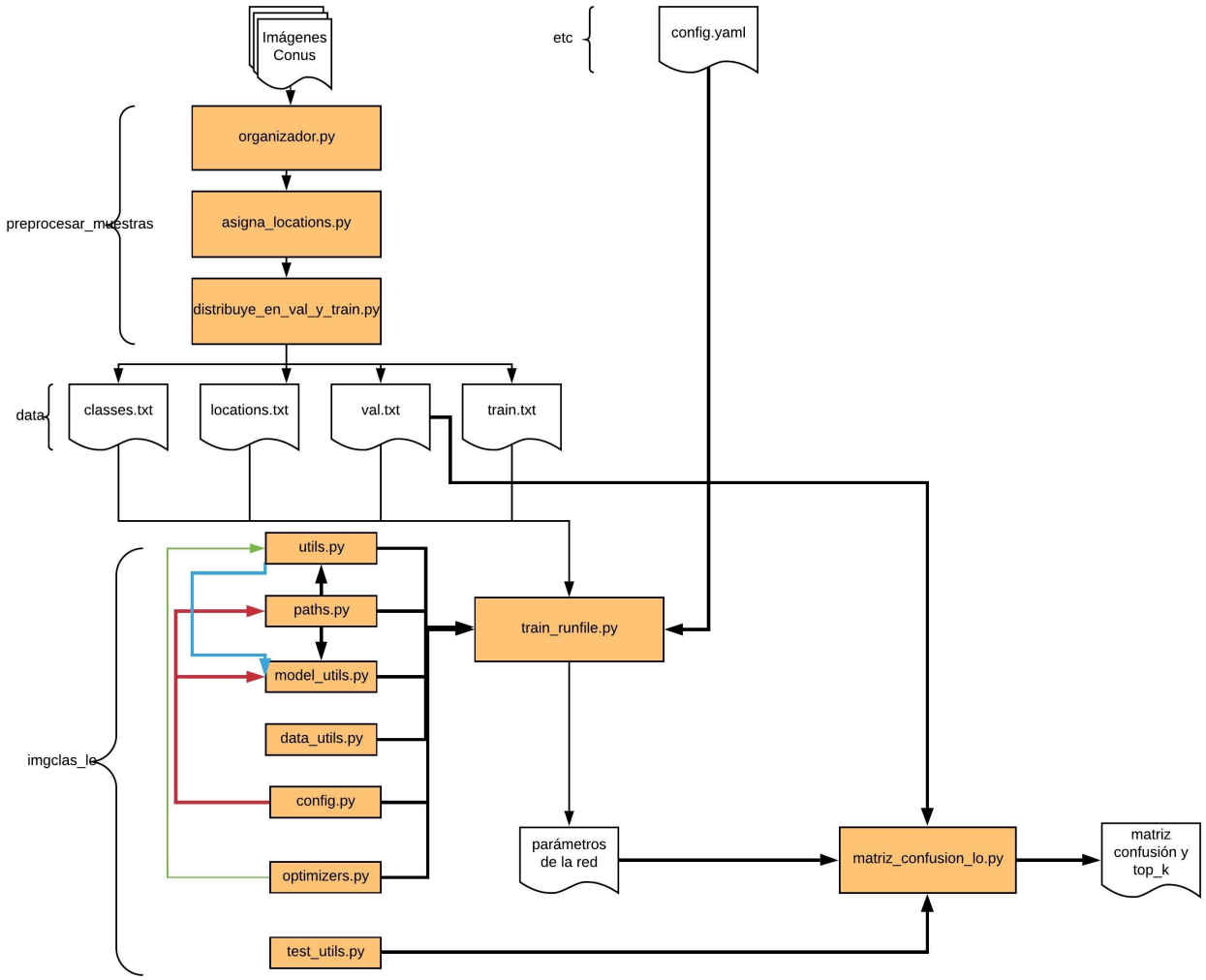

Figura 16. Diagrama que explica el funcionamiento del código implementado.

3.1 Preparación de las muestras

El código necesario se encuentra en el fichero llamado “preprocesar_muestras”.
En primer lugar, se ha desarrollado un script de Python que va recorriendo el fichero con los Conus. Se llama “organizador.py”. Lista la ubicación de cada imágen en el Docker y añade al final una etiqueta. Esto permitirá a la red saber dónde se encuentran las imágenes que necesita para entrenar. La etiqueta es un número que indica de qué especie se trata. Permitirá a la red indicar el nombre de la especie en vez del número de la etiqueta al predecir. Guarda todo esto en un fichero de texto llamado “muestras.txt”. Un ejemplo de una línea es el siguiente:

/srv/image-classification-tf/data/images/Conus//bruguieresi_SEN/010-bruguieresi-EA-north_of_dakar_almadies.jpg 46

Después, genera una leyenda para las eitquetas en otro fichero llamado “classes.txt”. Por ejemplo, la línea 46 del fichero “classes.txt” será “bruguieresi”.

Después, se aplica el script “asigna_locations.py” a “muestras.txt”. Recorre cada una de las extensiones que indican la localización de la imagen en el Docker y busca en este el nombre de la localización geográfica. Añade al final de dicha línea una nueva etiqueta, que ahora se corresponde con dicho área geográfica.

Continuando con el ejemplo, ahora en “muestras.txt” se tendrá:

```
/srv/image-classification-tf/data/images/Conus//bruguieresi_SEN/010-bruguieresi-EA-north_of_dakar_almadies.jpg 46 1
```

Donde 1 hace referencia al Atlántico Este, su localización.
Genera también un fichero con la leyenda de las localizaciones. Se llama “locations.txt”. Su primera línea será:

```
EASTERN_ATLANTIC
```

ya que es la localización geográfica del ejemplo anterior. Esta parte sólo es necesaria cuando usamos localizaciones para mejorar la precisión de la red.

Como último paso del preprocesado, se realiza una ordenación aleatoria de las rutas incluídas en “muestras.txt” . Se asigna un 90% a entrenamiento y el 10% restante a validación. Se almacenan en unos ficheros llamados “train.txt” y “val.txt”. Serán los que se le pasen a la red, junto con “locations.txt” y “classes.txt”. Esto lo realiza el script “distribuye_en_val_y_train.py”.

Tras todo esto, las muestras ya están preprocesadas y listas para introducirse en la red.

Se ha escrito también un programa que permite obtener los histogramas de distribución de las muestras de las distintas especies entre validación y entrenamiento. Son la figuras 4 y 5. Se llama “distribucion.py”

3.2 Entrenamiento y arquitectura del código desarrollado

Primero, se ha entrenado la red sin utilizar etiquetas de localización. Se han realizado 50 épocas con *batch size* de tamaño 16. Las entradas son las etiquetas de especie y las imágenes con las que se corresponden.

Se ha empleado la arquitectura Xception[10]. Se trata de un conjunto de 36 capas convolucionales. Combina activaciones *ReLU*, como las dadas por (5) en la introducción, con MaxPooling (explicada en 1.5). Para pasar a las capas densas se utiliza la activación *Max Average Pooling* (explicada en (1.5). Le sigue una capa densa con activación *ReLU* que retorna un vector de 1024 elementos.
Después de esto, se tiene una segunda capa densa. Es la de salida. Usa una activación de tipo *Softmax* como la dada por la ecuación (6) en la introducción. Se tiene así como salida de la red una distribución de probabilidad discreta y normalizada a la unidad para cada muestra del batch como la de (7). Formalmente es una matriz de tamaño (16,83). La primera componente de la fila i-ésima se corresponde con la probabilidad de que la imágen i-esima del batch que le hemos pasado pertenezca a la primera especie que aparece en “classes.txt” y así sucesivamente.
Como función de pérdida se ha usado *Categorical Cross-Entropy Loss*. Viene dada por la ecuación (8).

Como particularidad de nuestro trabajo, el tensor de salida tras aplicar todas las capas convolucionales tenía forma (2048,7,7). Visualmente se puede interpretar como 2048 matrices de tamaño (7,7). Tras aplicar *Max Average Pooling* se tiene un único vector de tamaño 2048. Cada coeficiente del vector es el promedio de todos los coeficientes de cada matriz (7,7). Toda la idea detrás de estos filtros se ha desarrollado en la sección 1.5.

El diagrama del modelo Xception se muestra en la figura 17 aportada por la fuente [10] de bibliografía.

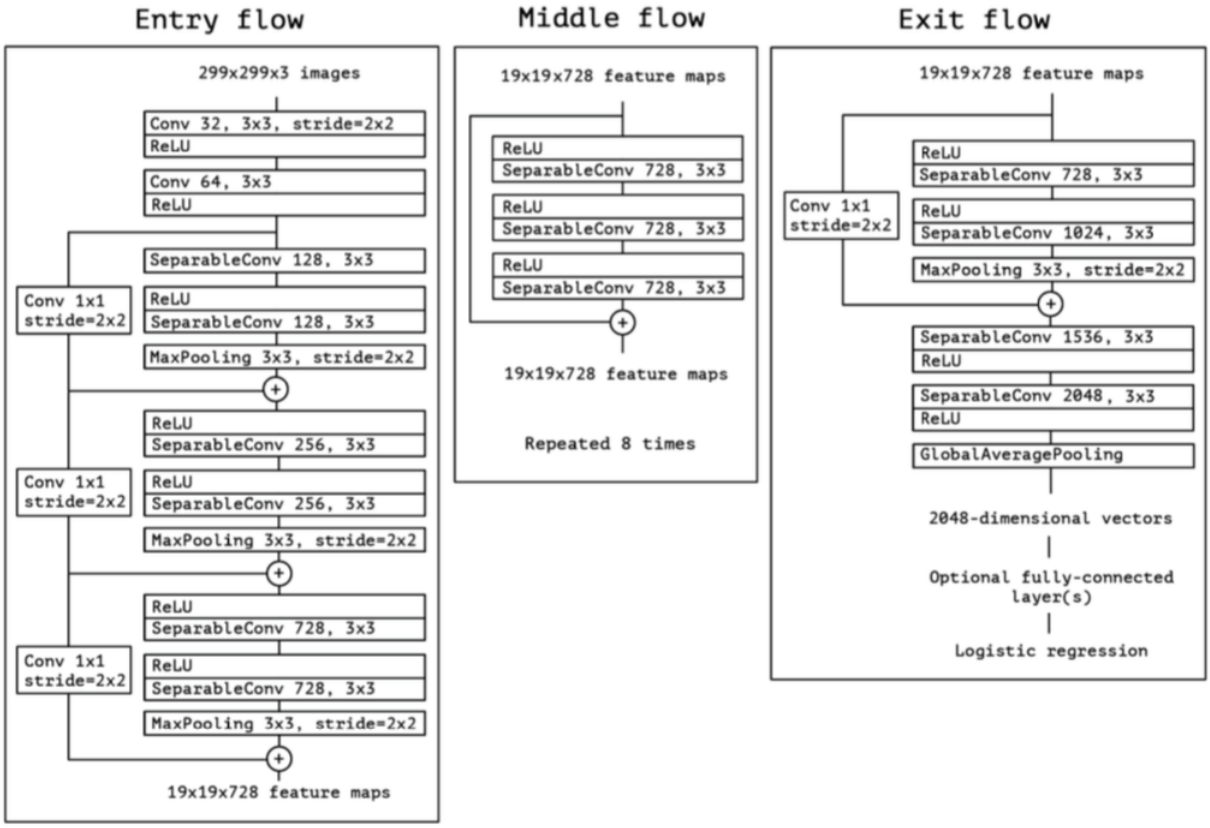

Figura 17. Modelo Xception. Aportado por referencia [10].

Como aumentación se han utilizado los valores dados en la figura 18. Se le pasan a “train_runfile.py” (el script principal de nuestro trabajo y que emplea todos los otros métodos para poder entrenar) desde:

/srv/image-classification-tf/config.yaml.

	Entrenamiento	Validación
Porcentaje máximo de giro vertical de la imágen	50 %	50 %
Porcentaje máximo de giro horizontal de la imagen	50 %	50 %
Frecuencia de rotación de la imágen	50 %	50 %
Ángulo máximo de rotación de la imágen	45º	45º
Porcentaje de zoom máximo	20 %	20 %
Ruido de píxeles artificial máximo	20 %	20 %

Figura 18. Datos de aumentación para las muestras de entrenamiento y validación. Son los rangos máximos. Cada vez que la red vea una nueva imágen, se le aplica aleatoriamente un valor dentro de estos rangos

Tras realizar el entrenamiento para las imágenes de Conus normalmente, se ha modificado la arquitectura del modelo para que admita como entrada adicional la etiqueta de localización por área geográfica.

Todo lo relacionado con el entrenamiento y predicción sin localizaciones se encuentra en la carpeta Imgclas. Se ha generado una nueva carpeta llamada Imgclas_lo para el código modificado que entrena con localizaciones. Además, será necesario incluir un fichero llamado “locations.txt” junto con el de “val.txt” y “train.txt” que ya usábamos. A estos se les tienen que añadir también etiquetas de localización. Se ha descrito en la sección 3.1. La carpeta con las imágenes de los Conus se encuentra en:

/srv/image-classification-tf/data/images

Dentro de “imgclas_lo”, nuevamente el script principal es “train_runfile.py”.Ahora, está modificado para entrenar con localizaciones. Esto ocurre con todos los otros métodos de este fichero. Principalmente, obtiene los tensores de entrada usando las imágenes de entrada, llama a “model_utils.py” para que cree el modelo, lo compila y lo entrena. Después, guarda los coeficientes de la red en:

/srv/image-classification-tf/data/models

Este método también ejecuta la API funcional para que saque por el puerto 6006 del ordenador las gráficas con la precisión acertando las especies y la función de pérdida L durante cada época del entrenamiento.

Entre los nuevas funciones de “imgclas_lo” destaca “create_model_lo”, perteneciente a model_utils.py. Este método crea el modelo que se va a entrenar para localizaciones. Carga los datos preentrenados de la red Xception y añade las capas densas a entrenar. Incluye esta línea tras realizar el *Max Average Pooling*:

```
Input_locations = Input(shape=(CONF[‘model’][‘num_locations'],))
```

Es aquí donde la red al entrenar va a necesitar que se le introduzcan las etiquetas de localización. Se deben pasar en el mismo orden en el que se han introducido las fotografías en el batch. Estas etiquetas de localización se añaden al final del vector obtenido al hacer *Max Average Pooling*. De este modo, la red densa tiene un parámetro más de entrada que ayuda a mejorar la precisión obtenida con *Softmax*.

Los tensores de las imágenes de entrada y sus etiquetas de localización y especie se introducen en el modelo mediante un generador. Se trata de una función que le entrega a la red las entradas necesarias durante cada época tras aplicarles aumentación. Se encuentra en “data_utils.py”.

Tras entrenar con localizaciones en el modelo antes descrito, se prueba a variar el tamaño de la última red densa. Se prueba a escoger 1024 en lugar de 2000 como tamaño del vector de salida. También se prueba a reducirlo a 600 componntes. Por último, se prueba a añadir una última capa densa antes de *Softmax* con un vector de salida de 600 componentes.

En la figura 19 se muestran los diagramas de los distintos modelos que se han realizado para clasificar los Conus.

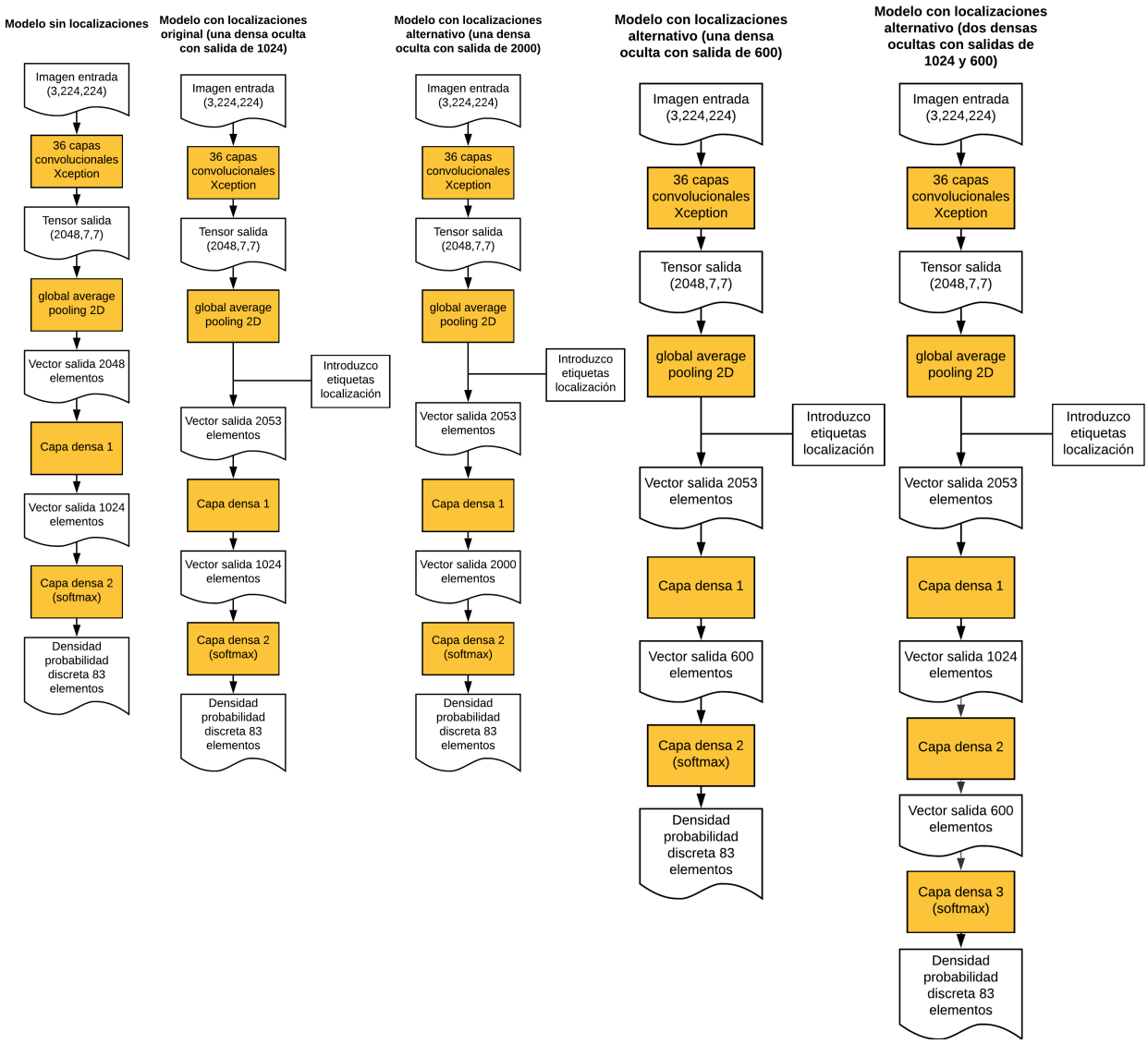

Figura 19. Diagrama de los diferentes modelos empleados. Se comparará las precisión obtenida por cada uno de ellos en las secciones 4 y 5.

3.3 Matriz de confusión

Tras entrenar, se genera la matriz de confusión para las muestras de validación y el *top_k* del modelo. Esto se realiza en “matriz_confusion.py” cuando no se usan localizaciones como datos adicionales para la predicción. En caso de que se estén usando se emplea “matriz_confusion_lo.py”. Ambas se encuentran en la carpeta “calcular_matrices_confusion”.

4. Resultados y análisis

4.1 Modelo sin localizaciones

Se ha entrenado la red siguiendo el modelo de la izquierda en la figura 19. No se han utilizado localizaciones. En las figuras 20 y 21 se muestra la precisión obtenida identificando las muestras de entrenamiento y validación durante cada una de las 50 épocas de entrenamiento. Se han usado 50 épocas con un *batch size* de 16 muestras en cada una. Las imágenes de entrada han tenido tamaño (3,224,224). Entre las capas convolucionales y la capa densa de salida con *Softmax* hay una capa densa que retorna un vector con 1024 coeficientes.

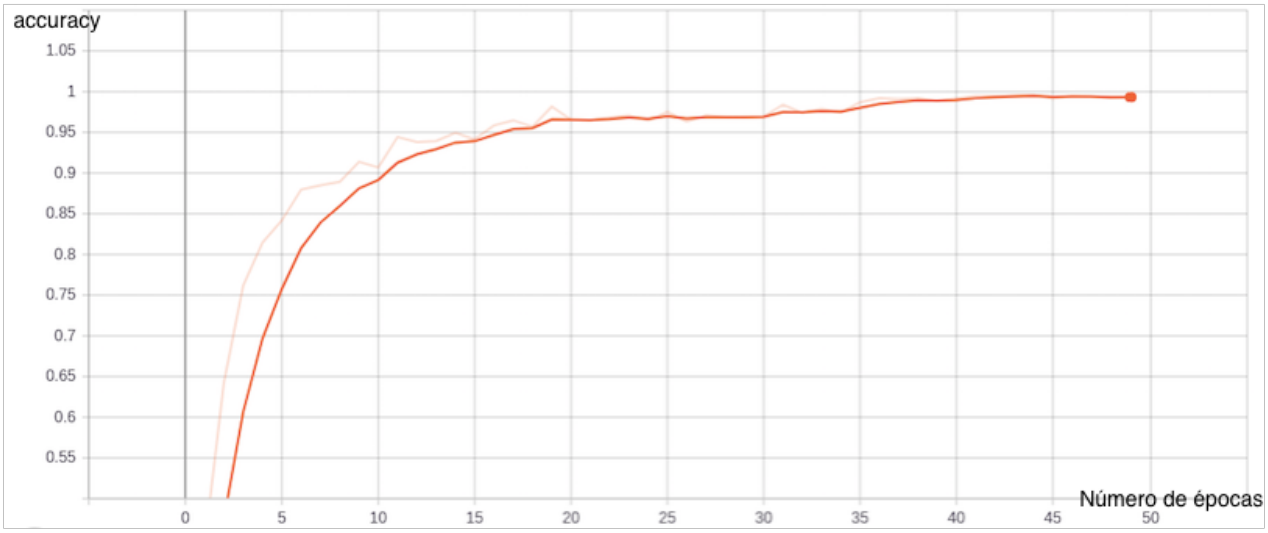

Figura 20. Evolución de la precisión de la red sin lozalizaciones distinguiendo Conus para las muestras de entrenamiento en cada una de las épocas de entrenamiento. Se trata del modelo original sin lozalizaciones en la época 50.

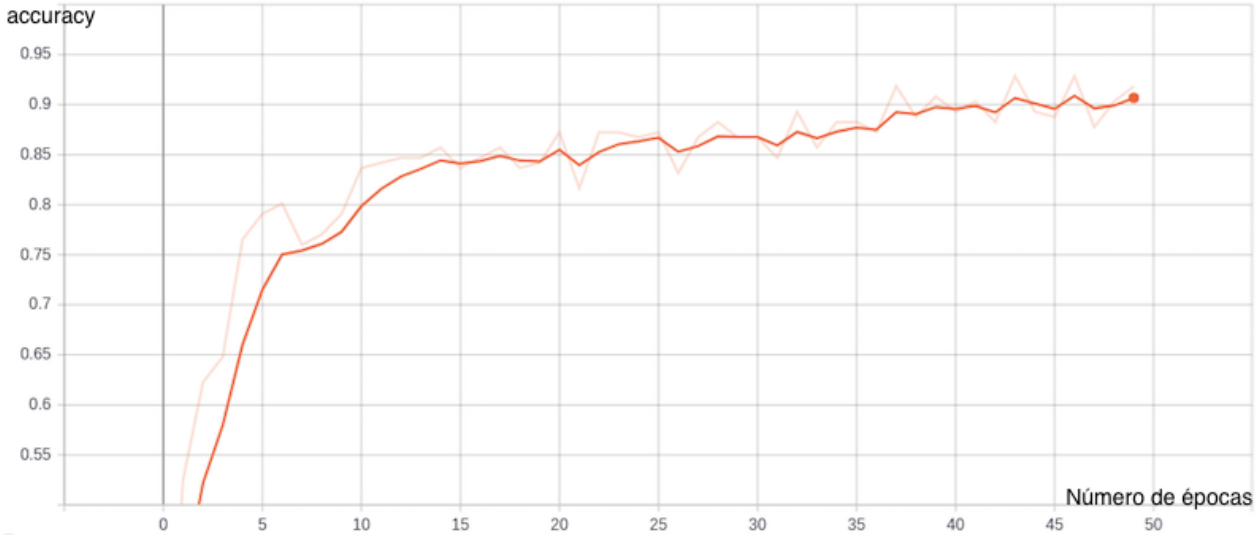

Figura 21. Evolución de la precisión de la red distinguiendo Conus para las muestras de validación tras cada época de entrenamiento. Se trata del modelo original sin localizaciones en la época 50.

Notamos que hasta la época 45, el aumento de precisión en las muestras de entrenamiento continúa aumentando significativamente. Los parámetros de la red aún siguen ajustándose a las muestras. La validación aumenta hasta entonces con valores, como es de esperar, inferiores a los de entrenamiento. En la figura 22 se muestra la evolución de la pérdida L_{CE} para ver que, en efecto, se va reduciendo. Notamos que hasta la época 40 se disminuye notablemente.

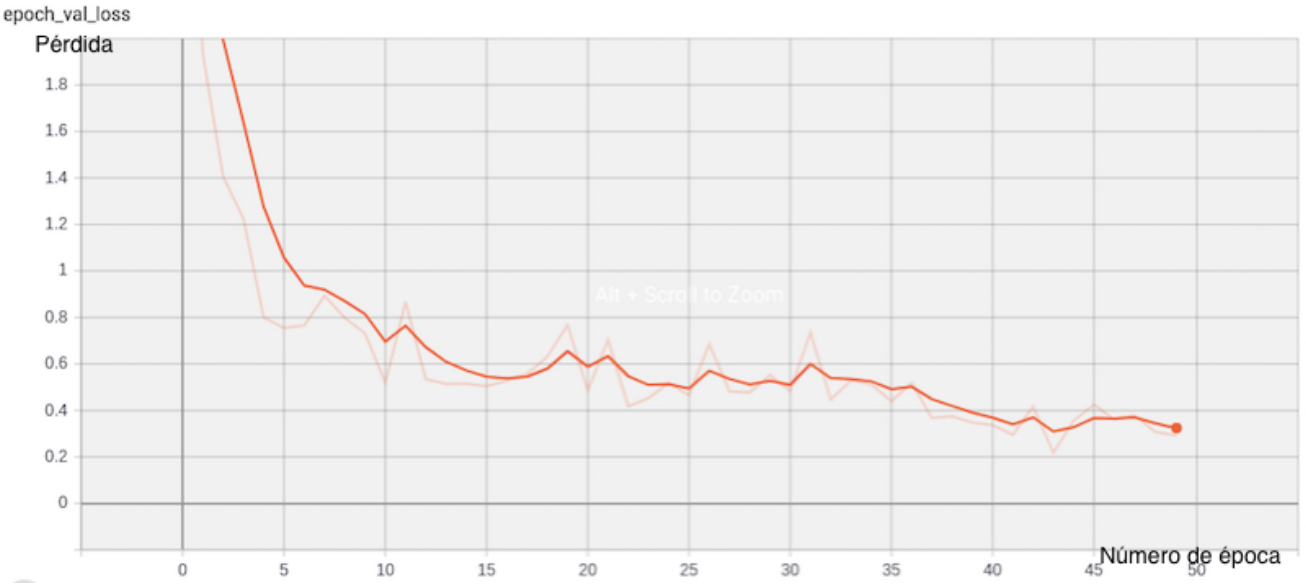

Figura 22. En el eje de ordenadas se muestra la pérdida L_{CE}. En el eje de abcisas, la época en la que ha tomado ese valor. Resultado obtenido para el modelo original que no emplea localizaciones. Tiene una capa densa oculta que retorna un vector de 1024 coeficientes.

La matriz de confusión obtenida con los parámetros de la época 50 para las muestras de validación se muestra en la figura 23.

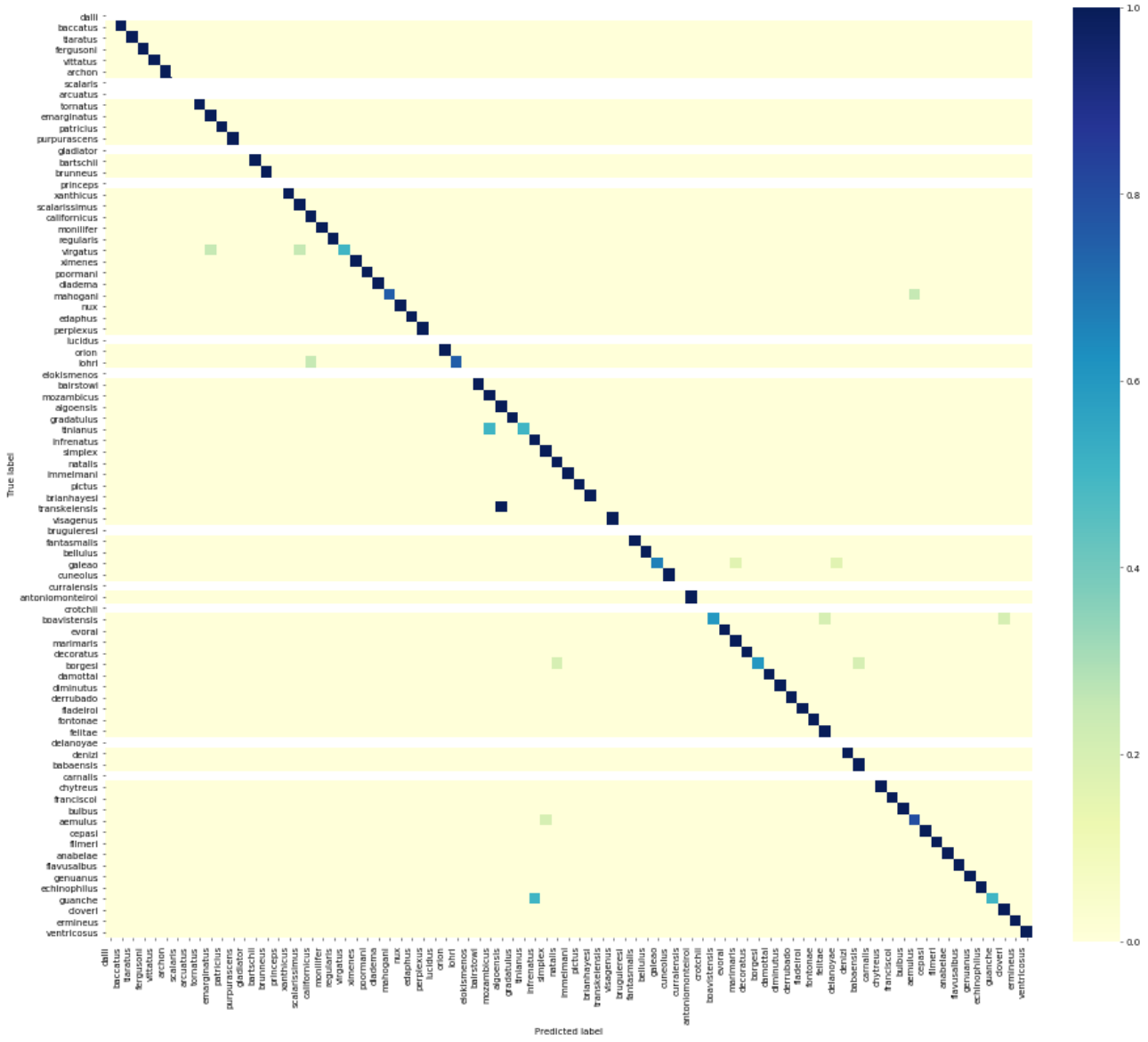

Figura 23. Matriz de confusión para la validación del modelo original sin localizaciones. Los coeficientes de la red tras son los obtenidos tras la época 50 de entrenamiento.

Como se ve, en general las predicciones más probables y las especies reales son las mismas salvo el caso de la especie *Transkeiensis* (localización indo pacífica). Si nos vamos al histograma de la figura 4, esta especie resulta ser una de las que tiene menos muestras para entrenar (la cuarta que menos). Probablemente se deba a esta falta de entradas para ajustar la red a dicha especie por lo que se tiene peor calidad en sus predicciones.
Si atendemos al *top_k* para validación se tiene la figura 24:

top_1/ %	92.9
top _2/ %	96.9
top_5/ %	100.0

Figura 24. Tabla con los *top_k* para las muestras de validación. Los coeficientes del modelo son los obtenidos para un 50 épocas de entrenamiento sin localizaciones.

Con los coeficientes obtenidos para nuestra red original sin localizaciones, el 92.9% de las ocasiones la especie que se ha considerado más probable para una muestra de validación ha sido la correcta.
El 96.9% de las ocasiones la especie correcta ha sido una de las dos más probables según la red.
El 100% de las ocasiones de las ocasiones, la especie correcta de las muestras de validación ha estado entre las 5 que la red ha considerado más probable.

Vamos a ver ahora cómo mejoran estas predicciones al añadir la opción de indicar la localización geográfica de las muestras.

4.2 Modelo original con localizaciones

Se ha entrenado la red siguiendo el segundo modelo de la figura 19. Se han añadido las localizaciones tras el global average pooling. Se tiene una red densa oculta antes de la de salida. Retorna un vector de 1024 coeficientes. En las figuras 25 y 26 se muestra la precisión obtenida identificando las muestras de entrenamiento y validación para cada una de las 50 épocas de entrenamiento.
Notamos que tarda algunas épocas menos en acercarse al 100% de precisión en las muestras de entrenamiento. El disponer de un parámetro de entrada adicional asociado a localizaciones le ha permitido aprender más rápido.

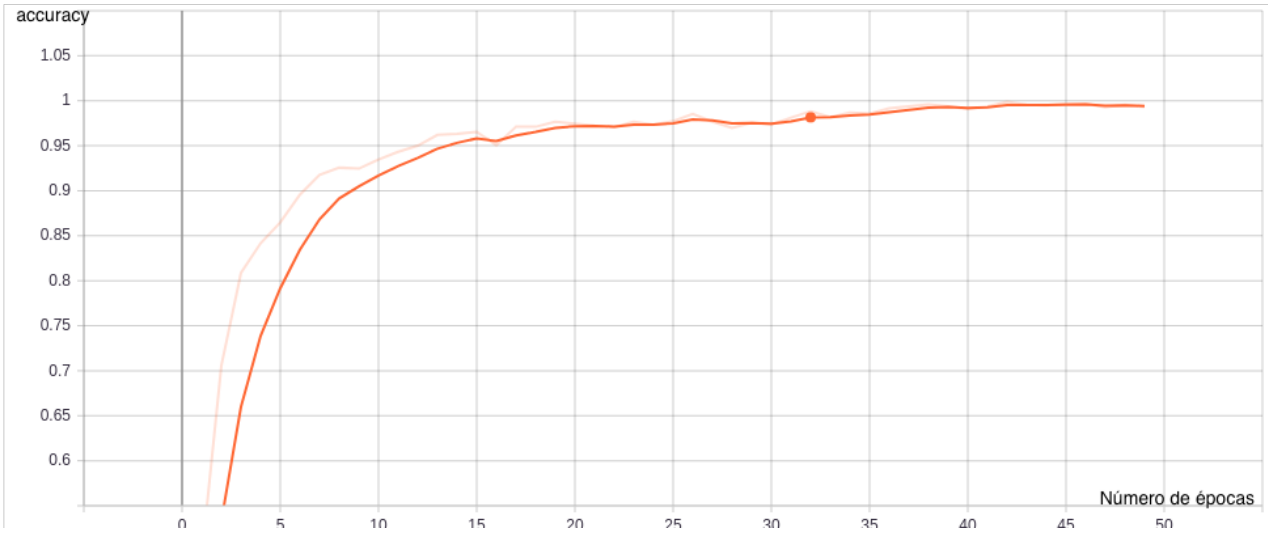

Figura 25. Precisión en las muestras de entrenamiento tras cada ciclo de aprendizaje de la red.

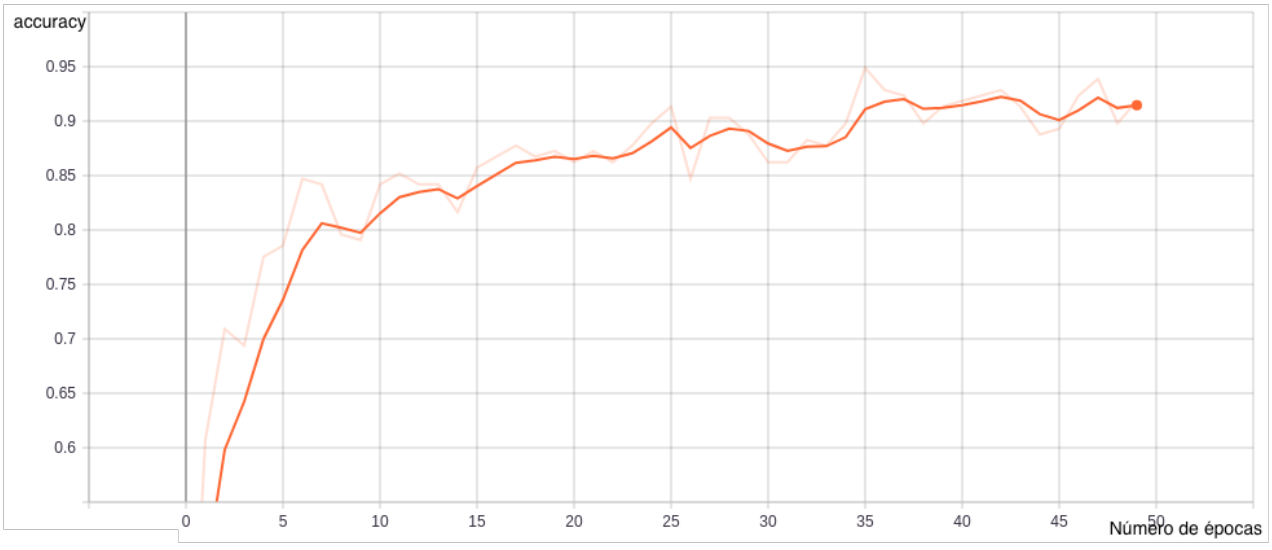

Figura 26. Precisión para predecir las muestras de validación en el modelo original al que se le han añadido etiquetas de localización geográfica.

Observamos que esta vez la precisión llega a un porcentaje algo superior para la validación. Tarda menos en superar el 90%. El uso de localizaciones ha mejorado el modelo. Sin embargo, a partir de la época 40 comienza a oscilar para validación. Esto se puede entender viendo la figura 25. Ya se ha alcanzado casi el 100% de precisión para las muestras de entrenamiento. La red ya ha "aprendido" los patrones para mapear esas muestras y empieza a sobreajustarse a dichos tensores de entrada.

En la figura 27 vemos la matriz de confusión para la validación con la red tras entrenar las 50 épocas. Notamos que el problema con las muestras de *Transkelensis* ya no está presente. Aunque se dispongan de menos muestras, el hecho de poder etiquetarlas por localización ha favorecido que se reconozcan mejor. Sigue teniendo algún caso en el que no identifica correctamente, pero ya no se repite tan frecuentemente para la misma muestra. Esto se sabe porque ninguna de estas casillas coloreadas que se salen de la diagonal toma una tonalidad oscura (correspondiente con que ha courrido varias veces).

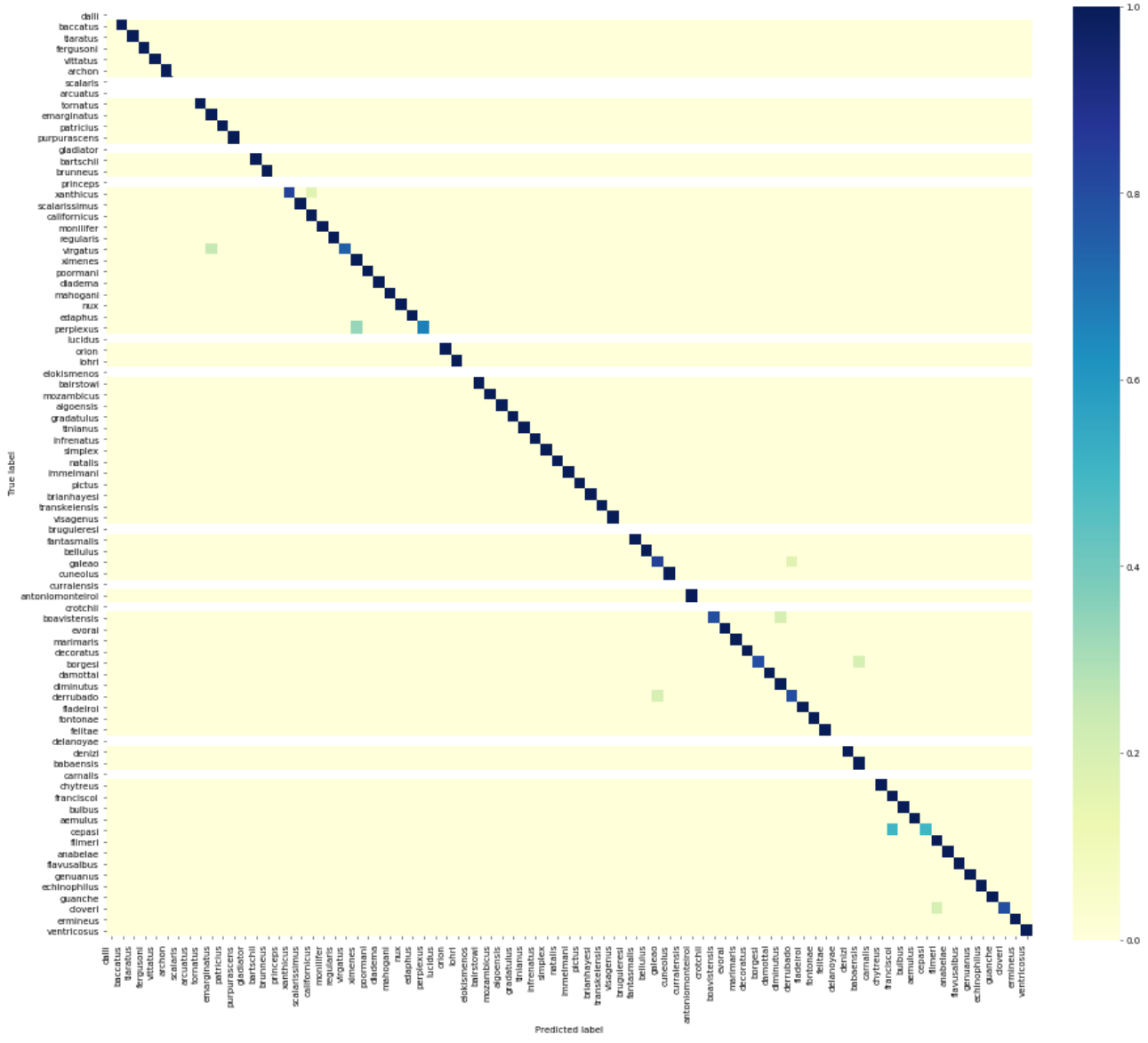

Figura 27. Matriz de confusión para la red original entrenada con localizaciones como dato de entrada adicional.

Es probable que si se dispusiera de más muestras de entrenamiento la matriz de confusión presentara aún mejores resultados. En la figura 28 se puede ver el *top_k*. Ahora se toma el *top_3* y *top_4* puesto que los modelos con localización tienen una precisión mayor y hay que tenerlos más en cuenta para compararlos.

top_1/ %	95.4
top_2/ %	98.5
top_3/ %	99.0
top_4/ %	100.0
top_5/ %	100.0

Figura 28. El *top_k* obtenido para las muestras de validación con 50 épocas. Se ha usado el modelo con localizaciones y una capa densa oculta que retorna un vector de 1024 coeficientes.

Nuevamente, notamos que el *top_1* y *top_2* son significativamente mayores a los obtenidos sin localizaciones.

Ahora se prueban las otras variantes de las capas densas. Todas ellas con localizaciones.

4.3 Modelo con localizaciones y modificando la parte densa

4.3.1 Red con capa densa que retorna vector de 600 coeficientes

Se ha cambiado la red densa que retornaba un vector de 1024 coeficientes por una donde sólo se retornan 600. Se ha reducido por tanto el tamaño del tensor W correspondiente. La red dispone de menos parámetros para aprender el modelo. Puede afectar positivamente reduciendo el sobreajuste o negativamente siendo insuficientes para aprender bien el problema. Lo comprobamos. En las figuras 29 y 30 se muestra la precisión de la red con las muestras de entrenamiento y validación durante el entreno.

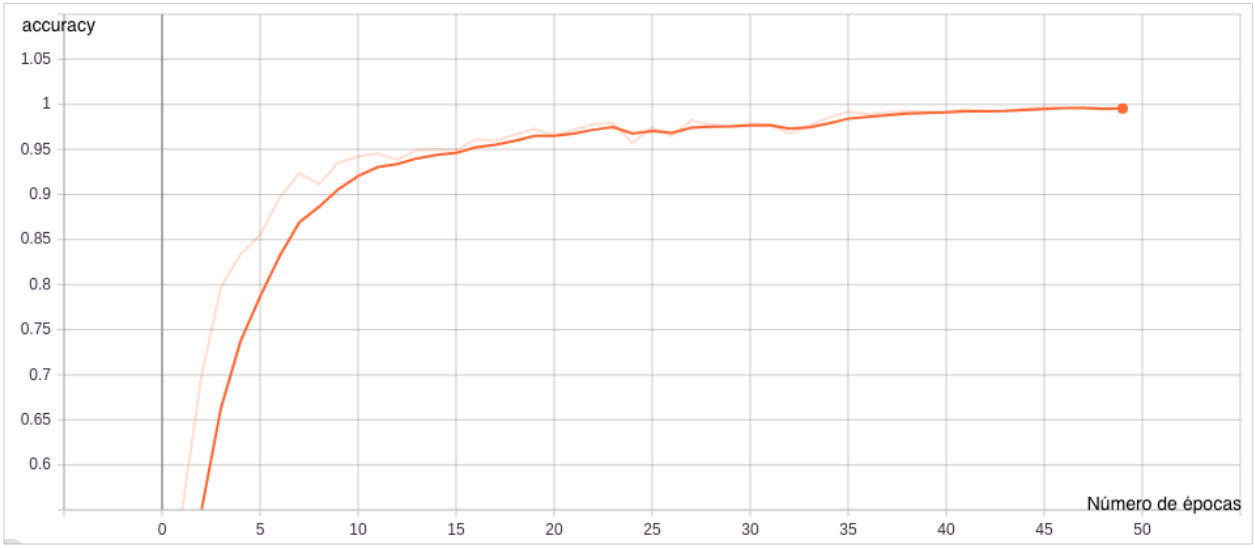

Figura 29. Precisión de la red con las muestras de entrenamiento en cada época. Se ha empleado el modelo con localizaciones y que tiene una red densa oculta que retorna un vector de 600 componentes.

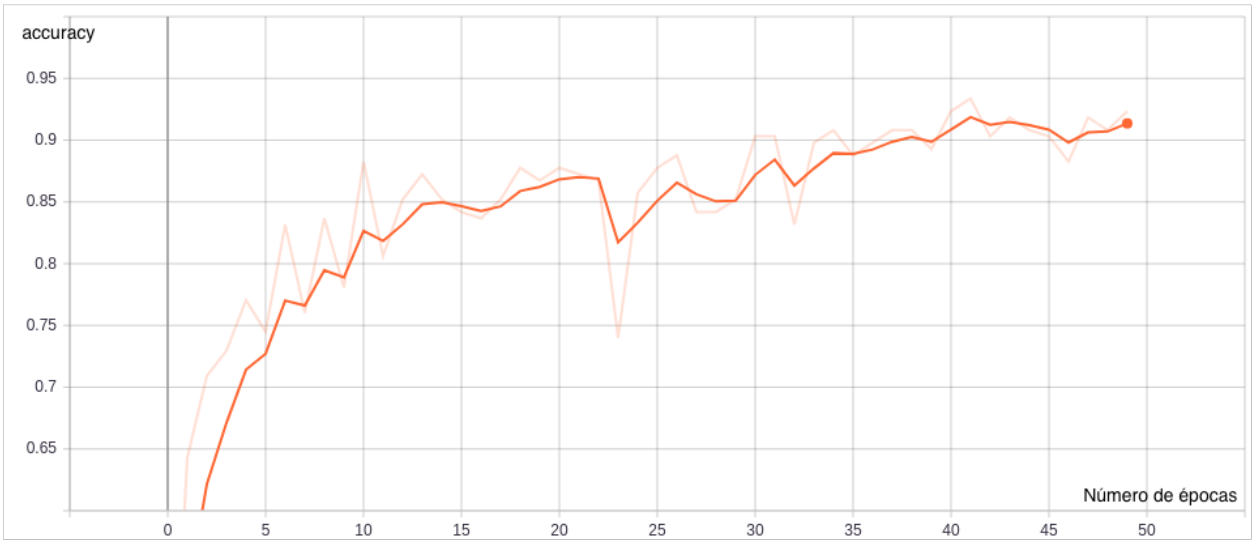

Figura 30. Precisión de la red con las muestras de validación en cada época.Se ha empleado el modelo con localizaciones y que tiene una red densa oculta que retorna un vector de 600 componentes.

Notamos que en este caso el mejor porcentaje para las muestras de validación se encuentra tomando 41 épocas. Notamos además que llega muy rápido al overfitting. No hay suficientes parámetros para ajustar tan bien como con el modelo de 1024. Además, coincide con la región en la que la red parece comenzar a sobreajustarse a las muestras de entrenamiento.
Comparamos los *top_k* con los parámetros de la época 41 y la final para decidir con cuál quedarnos. Se muestra en la figura 31.

	Época 41	**Época 50**
top_1/ %	93.9	93.4
top_2/ %	96.9	95.9
top_3/ %	98.5	98.0
top_4/ %	99.0	98.5
top_5/ %	99.0	98.5

Figura 31. El *top_k* para para las muestras de validación en el modelo con localizaciones y una densa intermedia que retorna un vector de 600 coeficientes. Los parámetros $W^{i}{}_{j}$ de la red tomando 41 épocas son mejores que en la final con 50 épocas.

En la época 41 los resultados son mejores que en la final. Sin embargo son notablemente inferiores que con la red densa de 1024 parámetros. Parece que no son suficientes parámetros libres para poder ajustar y aprender el las especies apropiadamante.
La matriz de confusión para la época 41 se muestra en la figura 32. Se observan claramente más errores que con la de 1024. Vuelve a aparecer el problema con la especie *Transkeiensis*. Hay “embotellamiento” de la información en la capa densa de 600 por falta de parámetros.

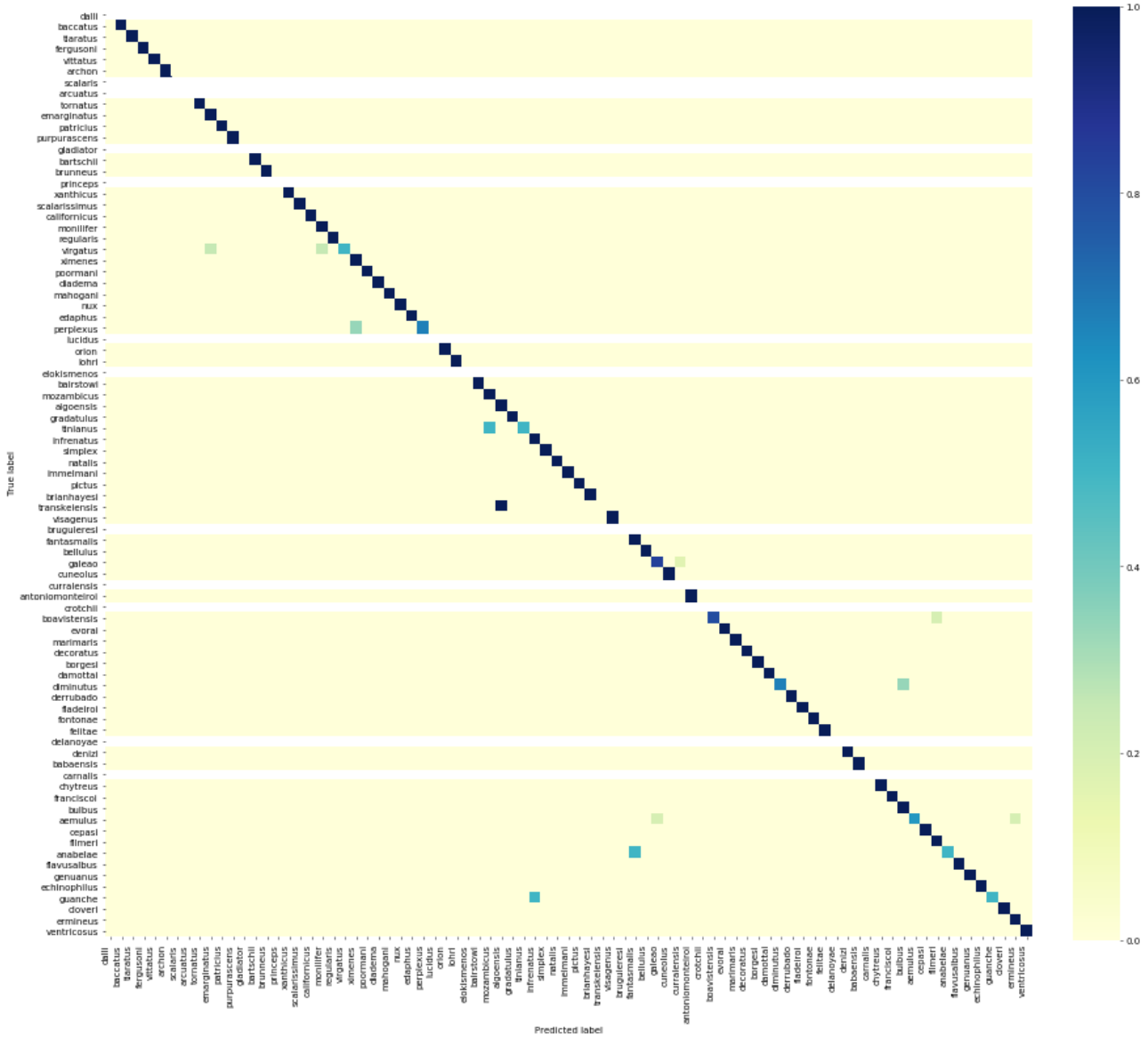

Figura 32. Matriz de confusión con las muestras de validación em el modelo con localizaciones y red densa que retorna un vector de 600 coeficientes. Es la época 41.

4.3.2 Red con capa densa que retorna vector de 2000 coeficientes

Se cambia ahora la red densa que retornaba 1024 originalmente por una que retorna 2000 coeficientes. Veremos si ayuda a aprender el modelo mejor o, por el contrario tantos parámetros para entrenar provoca un sobreajuste a los tensores de las imágenes de entrenamiento. Esto provocaría pérdida de generalidad y menor precisión en las muestras de validación.

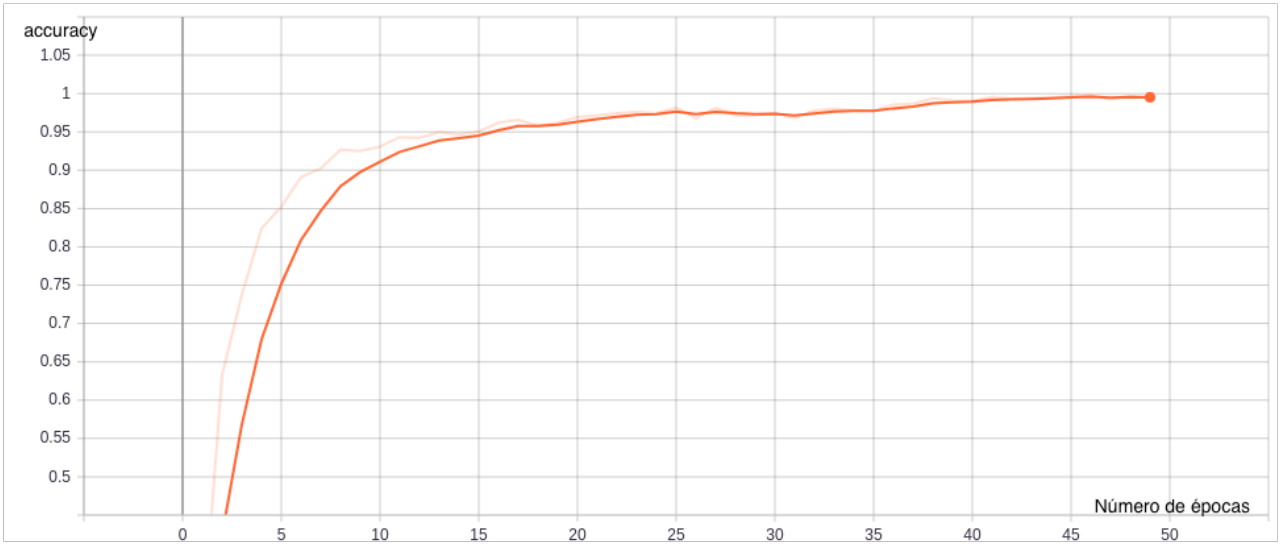

Figura 33. Precisión durante el entrenamiento del modelo con localizaciones y red densa de 2000 empleando las muestras de entrenamiento.

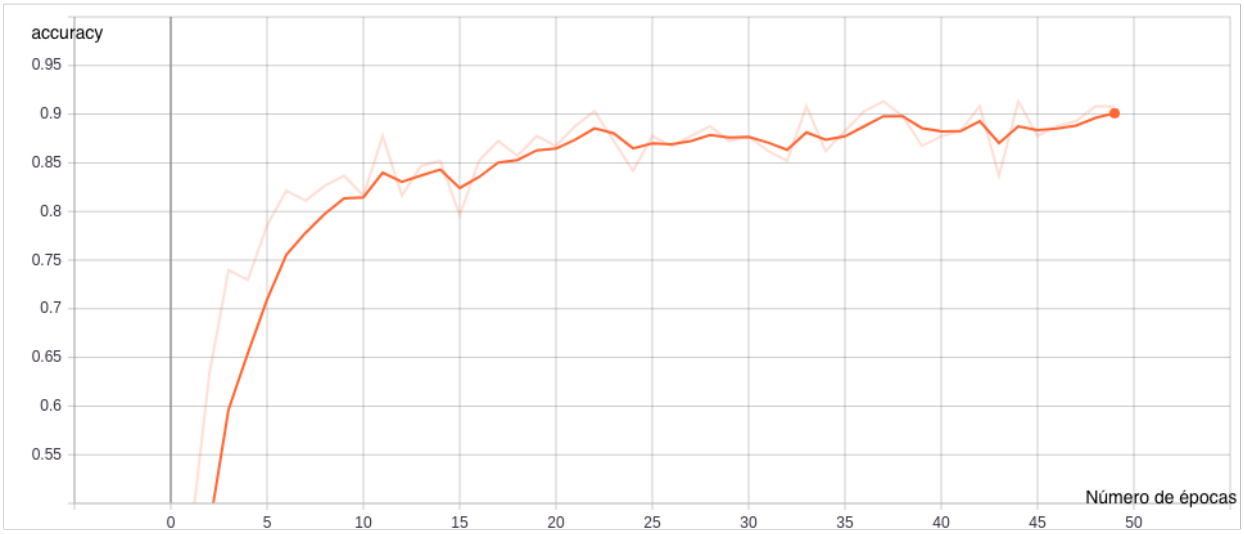

Figura 34. Precisión con las muestras de validación durante el entrenamiento del modelo con localizaciones y red densa de 2000.

En la figura 33 no parece notarse nada que nos haga pensar que el modelo vaya a ser mejor o peor que el de la red 1024. Sin embargo, en la validación vemos que la precisión apenas llega al 90%. Parece que en efecto, al haber tantos parámetros por ajustar, la red se está volviendo muy específica a las imágenes de entrada y pierde generalidad. Comparamos el *top_k* de la época 50 y el de la 38 ya que parece arrojar resultados similares. Sin embargo, si tomamos el de la 38 evitamos todo el overfitting adicional de las épocas posteriores donde ya se ha alcanzado una precisión próxima al 100% para las muestras de entrenamiento.

	Época 38	Época 50
top_1/ %	91.8	90.8
top_2/ %	96.4	94.4
top_3/ %	96.7	98.5
top_4/ %	99.0	99.5
top_5/ %	99.0	99.5

Figura 35. El *top_k* para las muestras de validación en el modelo con localizaciones y red densa de 2000.

Cuando tomamos un mayor número de candidatas a especie correcta por la red la época 50 da mejores resultados. Sin embargo, para el *top_1* y el *top_2* la época 38 es claramente mejor. Al final esto es más relevante ya que se cumpla el *top_1* implica directamente que la red ha acertado al predecir. Nos quedamos con los coeficientes $W^{i}{}_{j}$ de la época 38. Su matriz de confusión para las muestas de validación se muestra en la figura 36. Notemos que los resultados son incluso inferiores a los de la red sin localizaciones. Se observa claramente un caso de sobreajuste.

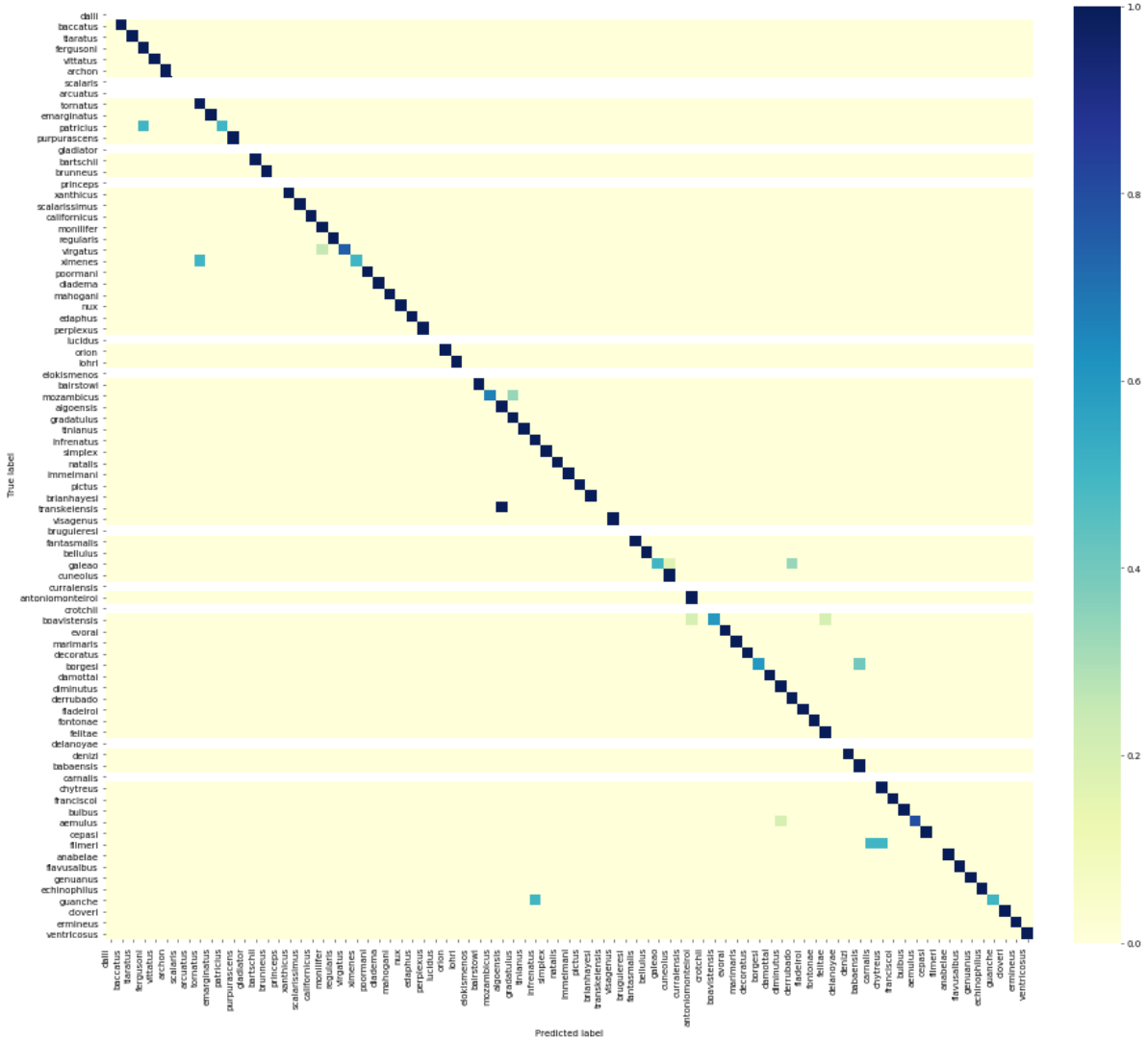

FIgura 36. Matriz de confusión para el modelo con localizaciones y red densa de 2000. Se observa una mayor cantidad de elementos fuera de la diagonal que con los otros planteamientos. Hay claro caso de sobreajuste.

4.3.3 Dos redes densas, retornan vectores de 1024 y 600 coeficientes

Por último se emplea el modelo de la derecha en la figura 19. Es como el original con localizaciones, pero se le añade una red densa adicional de 600.

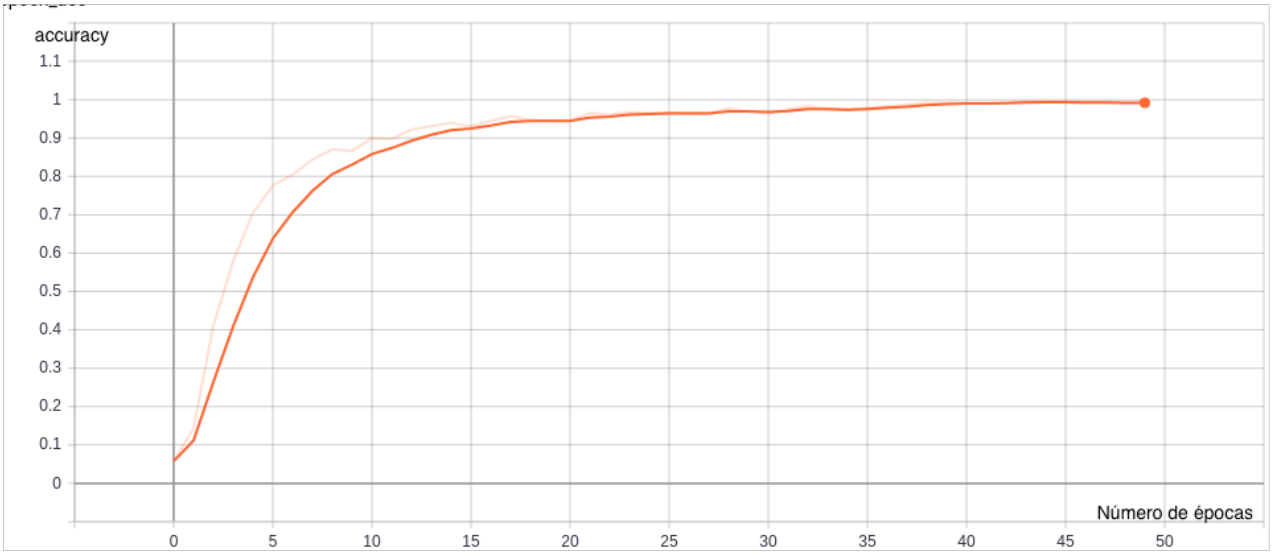

Figura 37. Precisión durante las épocas de entrenamiento para el modelo con localizaciones y dos capas densas ocultas.

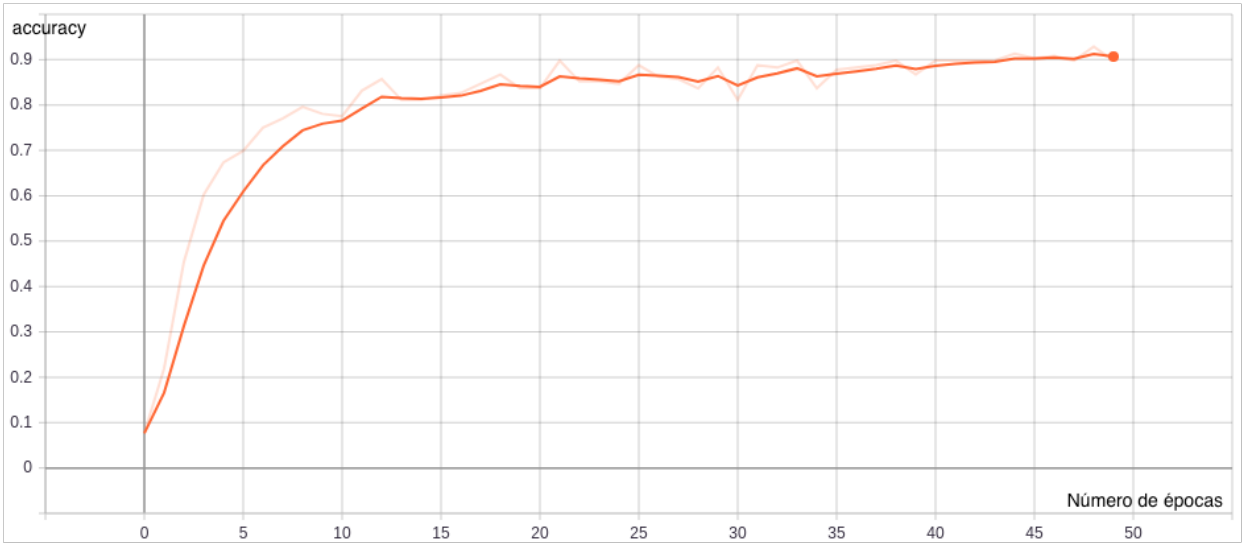

Figura 38. Precisión en las muestras de validación durante el entrenamiento de la red para el modelo con localizaciones y dos capas densas ocultas.

Notamos que la validación es menor que teniendo una sola capa densa de 1024. Parece que la reducción de parámetros previa la capa de salida provoca nuevamente un “embotellamiento” de la información. Faltan parámetros para no perder información valiosa para distinguir las especies.
En la figura 39 comparamos el *top_k* de validación tras 45 y 50 épocas. Desde la época 45 las muestras de entrenamiento se han calculado casi con un 100% de precisión. La mínima mejora que vemos en validación es casualidad, ya que la red no tiene mucha más información que aprender.

	Época 45	Época 50
top_1/ %	92.9	92.9
top_2/ %	95.9	97.4
top_3/ %	98.5	98.0
top_4/ %	98.5	98.5
top_5/ %	98.5	98.5

Figura 39. El *top_k* con las muestras de validación para el modelo con localizaciones y dos capas densas intermedias de 1024 y 600.

Notamos que se ha obtenido un *top_2* notablemente mayor en el caso de las 50 épocas. Sin emargo, sigue siendo un modelo peor que el de la red con 1024 solamente.

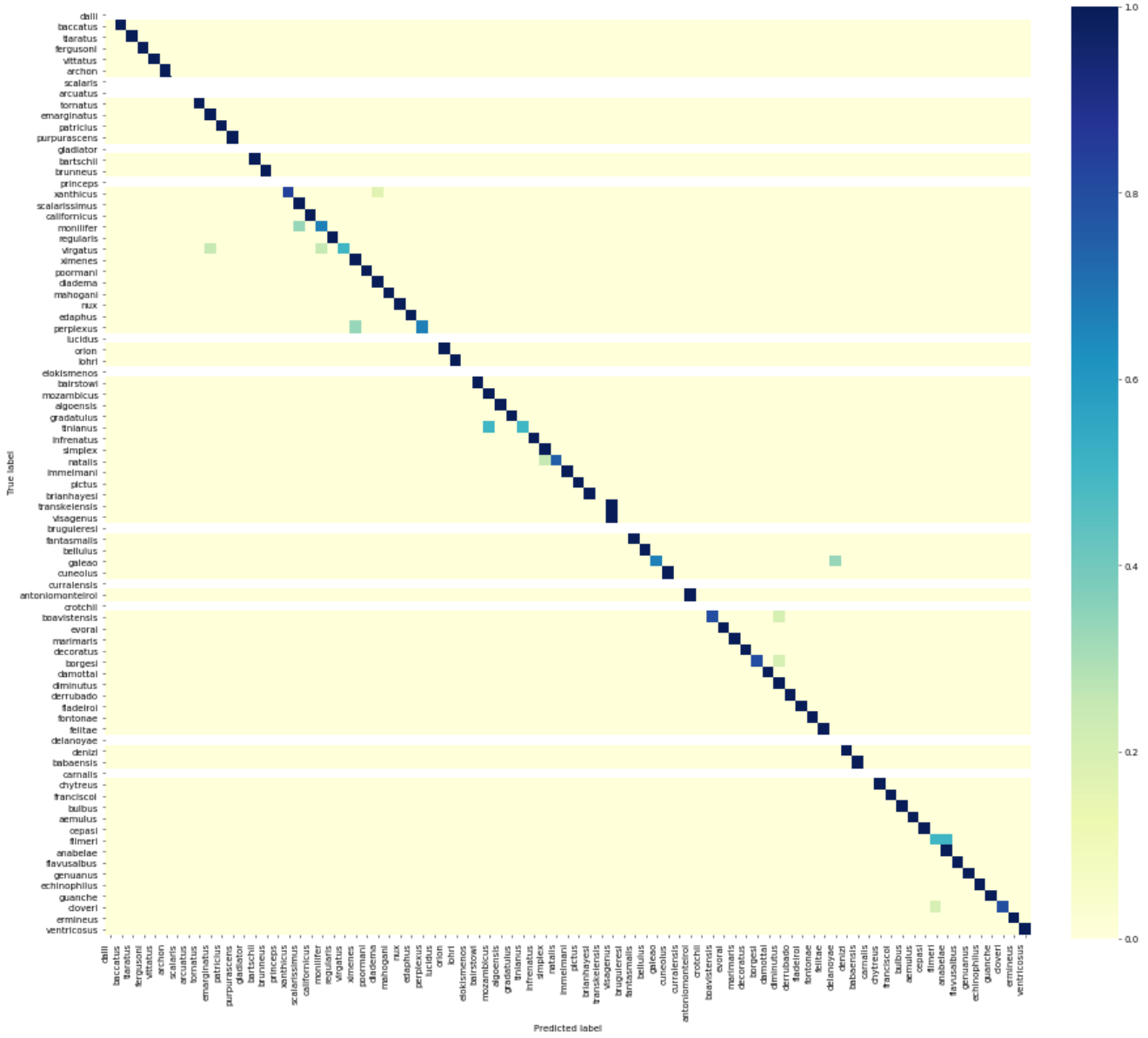

Figura 40. Matriz de confusión para la época 50 del modelo con dos capas densas.

5. Conclusiones y discusión

Se ha obtenido un precisión del 92.9% para la distinción de muestras de Conus por especie empleando la red convolucional Xception junto a un *Max Average Pooling*, una capa densa que retorna 1024 parámetros y la activación de probabilidad discreta *Softmax*. La pérdida se ha calculado usando *Categorical Cross-Entropy Loss.*

Se ha mejorado el modelo añadiendo localizaciones. Esto ha llevado a una precisión del 95.4% respetando las capas de la red original.

Cuando se han modificado el número de coeficientes, la precisión de la red ha decaído. Esto es debido al exceso de los mismos (sobreajuste) o al defecto (embotellamiento).

Se observa que la precisión del modelo depende notablemente del planteamiento del mismo. Sin embargo, la precisión podría aumentar si se dispusiera de más muestras para entrenar. En ese caso, tal vez los 2000 coeficientes en la capa densa no fueran tan excesivos.
Se podría, en lugar de realizar *Max Average Pooling*, promediando los coeficientes de las matrices de salida de la red convolucional, habe reempleado *Flatten* (se conservan todos los coeficientes). Algunos artículos recientes de vanguardia en el campo de la visión artificial muestran resultados favorables con estos [9]. Sin embargo, tal cantidad de coeficientes es excesiva para la memoria de la GPU empleada. Sería necesario emplear varias GPUs.

Durante los últimos días previos a la entrega del TFG, el Museo Nacional de Ciencias Naturales ha sugerido la posibilidad de añadir una clasificación más. Atendería a estudios mitogenómicos. Aún no están disponibles dichos resultados, pero si esto añadiera una etiqueta de entrada adicional más específica que la localización geográfica (sólo hay 5 distintas), los resultados podrían mejorar. Parece razonable asumir que, conociendo más detalles y más concretos de cada muestra, se facilite su clasificación.

6. Bibliografía

[1] Dan Guest, Kyle Cranmer y Daniel Whiteson. Deep Learning and its application to LHC physics. Annual Reviews. Junio 2018.

[2] Página consultada para entender la arquitectura de Von Neumann.
URL: https://www.sciencedirect.com/topics/computer-science/von-neumann-architecture. Visitada en mayo de 2020.

[3] François Chollet. Deep Learning with Python. Ed Manning. Enero 2018.

[4] José G. Delgado-Frías y William R. Moore. VLSI for neural networks and artificial intelligence. 2012.

[5] Página con información divulgativa sobre el funcionamiento de los distintos componentes del ordenador. URL: https://www.adslzone.net/2018/01/12/cpu-vs-gpu-diferencias . Visitada en febrero de 2020.

[6] Explicación detallada de la activación *ReLU* y en qué modelos ofrece mejores resultados.
URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ . Visitada en mayo de 2020.

[7] Página de github de Raúl Gómez Bruballa, investigador de la Universidad Autónoma de Barcelona especializado en Camputer Vision. Detalla el uso y la fórmula de la función de pérdida *Categorical Cross-Entropy Loss*.
URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/ . Consultada en Junio de 2020.

[8] Web con toda la documentación de la librería Keras. URL: https://keras.io .

[9] Hilo de un proyecto empleando Keras en el que uno de los desarrolladores de dicha librería explica las ventajas de *Flatten* en modelos entrenados previamente.
URL: https://github.com/keras-team/keras/issues/8470 .

[10] F.Chollet. Xception: Deep Learning with DepthWise Separable Convolutions. Abril 2017.

[11] Enlace a mi repositorio GitHub, donde se encuentra subido todo el código de las prácticas de Python realizadas previamente al TFG.
URL: https://github.com/lmc00/practicas_python

[12] Proyecto del grupo de computación avanzada y e-Ciencia del IFCA. Contiene una imagen de Docker descargable con una red ya preentrenada para distinguir imágenes genéricas. Visitada en diciembre de 2019.
URL: https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-image-classification-tf.html

[13] Repositorio de github donde está subido el Docker con todo mi trabajo. Es posible descargarlo y ejecutarlo. También se puede leer desde ahí todo el código de Python del TFG.
URL: https://github.com/lmc00/TFG .

[14] Enlace a la base de datos Imagenet. Consultado en Junio de 2020.
URL: http://www.image-net.org/ .